Tampere University

Joona Luoma

MULTI-TENANT HYBRID CLOUD ARCHITECTURE

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
September 2019

ABSTRACT

Joona Luoma: Multi-tenant hybrid cloud architecture
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
September 2019

This paper examines the challenges associated with *the multi-tenant hybrid cloud architecture* and describes how this architectural approach was applied in two software development projects. The motivation for using this architectural approach is to allow developing new features on top of monolithic legacy systems – that are still in production use – but without using legacy technologies. The architectural approach considers these legacy systems as *master systems* that can be extended with multi-tenant cloud-based *add-on applications*. In general, legacy systems are run in customer-operated environments, whereas add-on applications can be deployed to cloud platforms. It is thus imperative to have a means connectivity between these environments over the internet. The technology stack used within the scope of this thesis is limited to the offering of the .NET Core ecosystem and Microsoft Azure.

In the first part of the thesis work, a literature review was carried out. The literature review focused on the challenges associated with the architectural approach, and as a result, a list of challenges was formed. This list was utilized in the software development projects of the second part of the thesis. It should be noted that there were very few high-quality papers available focusing exactly on the multi-tenant hybrid cloud architecture, so, in the end, source material for the review was searched separately for multi-tenant and for hybrid cloud design challenges. This factor is noted in the evaluation of the review.

In the second part of the thesis work, the architectural approach was applied in two software development projects. Goals were set for the architectural approach: the add-on applications should be developed with modern technology stacks; their delivery should be automated; their subscription should be straightforward for customer organizations and they should leverage multi-tenant resource sharing.

In the first project a data quality management tool was developed on top of a legacy dealership management system. Due to database connectivity challenges, confidentiality of customer data and authentication requirements, the implemented solution does not fully utilize the architectural approach, as having the add-on application hosted in the customer environment was the most reasonable solution. Despite this, the add-on application was developed with a modern technology stack and its delivery is automated. The subscription process does involve certain manual steps and, if the customer infrastructure changes over time, these steps must be repeated by the developers. This decreases the scalability of the overall delivery model.

In the second project a PDA application was developed on top of a legacy vehicle maintenance tire hotel system. The final implementation fully utilizes the architectural approach. Support for multi-tenancy was implemented using ASP.NET Core Dependency Injection and Finbuckle.MultiTenancy-library. Azure Relay Hybrid Connection was used for hybrid cloud connectivity between the add-on application and the master system. The delivery model incorporates the same challenges regarding subscription and customer infrastructure changes as the delivery model of the data quality management tool. However, the manual steps associated with these challenges must be performed only once per customer – not once per customer per application. In addition, the delivery model could be improved to support customer self-service governance, enabling the delegation of any customer environment installations to the customers themselves. Even further, the customer environment installation could potentially cover an entire product family. As an example, instead of just providing access for the PDA application, the installation could provide access for all vehicle maintenance family add-on applications. This would make customer environment management easier and developing new add-on applications faster.

Keywords: multi-tenant, hybrid cloud, SaaS, Azure, .NET

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Joona Luoma: Moniosapuolihybridipilviarkkitehtuuri
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Syyskuu 2019

Tässä diplomityössä tutkitaan moniosapuolihybridipilviarkkitehtuuriin liittyviä haasteita ja kuvataan arkkitehtuurimallin soveltamista kahdessa ohjelmistokehityshankkeessa. Arkkitehtuurimallin soveltamisen varsinainen syy on tarve jatkaa pitkään kehitettyjen, monella asiakasyrityksellä edelleen käytössä olevien perintöjärjestelmien toiminnallisuuksien kehittämistä ilman, että kehitystä tarvitsee suorittaa perintöteknologioilla suoraan perintöjärjestelmään. Arkkitehtuurimallissa perintöjärjestelmä nähdään *pääjärjestelmänä*, jonka oheen voidaan kehittää moniosapuolisia pilvipohjaisia *lisäarvojärjestelmiä*. Perintöjärjestelmää ajetaan tyypillisesti asiakkaan omassa ympäristössä, kun taas lisäarvojärjestelmiä on tarkoitus ajaa pilvipohjaisesti, mikä luo tarpeen näiden ympäristöjen yhteen liittämiselle hybridipilviteknologioin.

Työn ensimmäisessä vaiheessa suoritettiin kirjallisuuskatsaus arkkitehtuurimallin haasteisiin, ja sen lopputuloksena saatiin tarkistuslistanomainen joukko haasteita, jota hyödynnettiin työn toisen vaiheen ohjelmistokehityshankkeissa. Huomionarvoista on, että kirjallisuudesta löytyy runsaasti tapausesimerkkejä liittyen erikseen moniosapuolisuus- ja hybridipilvisovelluksiin, mutta täsmälleen moniosapuolihybridipilviarkkitehtuurimalliin liittyen aiempia laadukkaita tutkimuksia ei löytynyt. Tästä syystä kirjallisuuskatsauksessa etsittiin lähdemateriaalia näihin kahteen menetelmään liittyen erikseen. Tämä seikka otetaan huomioon katsauksen tuloksen arvioinnissa.

Työn toisessa vaiheessa arkkitehtuurimallia sovellettiin kahdessa kehityshankkeessa. Kehityshankkeita varten arkkitehtuurimallin soveltamiselle asetettiin tavoitteita, joiden toteutumista arvioidaan kunkin kehityshankkeen arviointiosioissa. Tavoitteena oli luoda lisäarvojärjestelmät modernilla teknologiapinolla; automatisoida lisäarvojärjestelmien toimittaminen; tehdä lisäarvojärjestelmien tilaamisesta asiakasyrityksille suoraviivaista ja hyödyntää pilviresurssien jakamista asiakasyritysten kesken moniosapuolisuuden avulla.

Ensimmäisessä kehitysprojektissa kehitettiin datan laadunhallintasovellus asiakastiedonhallintapääjärjestelmän oheen. Asiakastiedon luottamuksellisuudesta ja tietokantayhteyteen sekä käyttäjien autentikointiin liittyvistä syistä johtuen arkkitehtuurimallia ei hyödynnetty tässä kehitysprojektissa kokonaisvaltaisesti. Kehitettyä lisäarvojärjestelmää ajetaan kunkin asiakkaan omassa ympäristössä, mutta se hyödyntää toimitusautomaatiosta, ja se on kehitetty nykyaikaisella teknologiapinolla. Lisäarvojärjestelmän tilaamien pitää sisällään manuaalisia toimitustoimenpiteitä, ja mahdolliset muutokset asiakkaan infrastruktuurissa voivat johtaa näiden manuaalisten toimitustoimenpiteiden toistamiseen. Nämä manuaaliset toimitustoimenpiteet vaativat kehittäjien panostusta ja ne vähentävät siksi toimitusprosessin skaalautuvuutta.

Toisessa kehitysprojektissa kehitettiin käsipäätesovellus ajoneuvohuollon rengashotellipääjärjestelmän oheen. Kehitetty lisäarvosovellus hyödyntää arkkitehtuurimallia. Sovelluksen moniosapuolisuus toteutettiin ASP.NET Core -verkkosovelluskehykseen sisäänrakennetun riippuvuusinjektiokontin ja Finbuckle.Multitenancy-kirjaston avulla. Azure Relay Hybrid Connection -palvelua hyödynnettiin hybridiyhteyden muodostamiseen lisäarvojärjestelmästä pääjärjestelmään. Sovelluksen toimitusmalli pitää sisällään samat tilaukseen ja asiakasympäristön infrastruktuurin muutoksiin liittyvät haasteet kuin ensiksi mainitun kehityshankkeen toimitusmalli. Haasteet ovat kuitenkin osin kertaluonteisia: niihin liittyvät mahdolliset manuaaliset toimenpiteet tulee suorittaa vain kerran kutakin asiakasta kohden, ei kerran kutakin asiakasta ja järjestelmää kohden. Järjestelmän toimitusmallia voidaan kuitenkin jatkokehittää sellaiseksi, että vastuu asiakasympäristöasennusten hallinnoinnista siirtyisi asiakkaan ylikäyttäjille palveluntarjoajan kehittäjien sijaan. Toimitusmallia on mahdollista jatkokehittää jopa tuoteperhetasoiseksi: esimerkiksi ajoneuvoliiketoiminnan laajuudella asiakasympäristöasennukset voitaisiin suorittaa vain kerran, mikä yksinkertaistaisi asiakasympäristöasennusten hallinnointia ja kehittämistä.

Avainsanat: moniosapuoli, hybridipilvi, SaaS, Azure, .NET

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

PREFACE

Writing this thesis has been an extraordinary experience: a true *grand final* for my educational journey. All ups and downs included, writing the paper, conducting the research and implementing the applications have taught me a lot about – not only cloud paradigms – but also about time and complexity management. I am happy with the final results and I have absolutely no doubt that the know-how gained during the process will be beneficial in oncoming endeavors too.

I would like to thank the examiner of this thesis, asst. prof. Davide Taibi, for providing valuable guidance especially in the early stages of the process. Finally, I would like to thank everyone who have been supporting me through the writing process: my family, my friends and my colleagues.

Tampere, 28 September 2019

Joona Luoma

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AACS	Azure App Configuration Service
AppConfig	ASP.NET Core Application Configuration
API	Application Programming Interface
ASP	Application Service Providing
CD	Continuous Delivery
CI	Continuous Integration
CRM	Customer Relationship Management
CSP	Cloud Service Provider
DevOps	Development Operations
DI	Dependency Injection
DMS	Dealership Management System
DQM	Data Quality Management
EAV	Entity-Attribute-Value
ERP	Enterprise Resource Planning
FaaS	Function as a Service
IaaS	Infrastructure as a Service
IIS	Internet Information Services
IP	Internet Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
LoT	Level of Tenancy
MTEL	Multi-Tenancy Enablement Layer
MTHC	Multi-Tenant Hybrid Cloud
NIST	the National Institute of Standards and Technology
NoSQL	Not only SQL
ODBC	Open Database Connectivity
on-prem	On-premises
OTS/ASP	Off-The-Shelf Application Service Providing
PaaS	Platform as a Service
PDA	Personal Digital Assistant
RDB	Relational Database
RDBMS	Relational Database Management System
SaaS	Software as a Service
SLA	Service Level Agreement
SMS	Systematic Mapping Study
SPL	Software Product Line
SQL	Structured Query Language
SSO	Single Sign-On
THPDA	Tire Hotel Personal Digital Assistant
THS	Tire Hotel System
URL	Uniform Resource Locator
VNET	Azure Virtual Network
VPN	Virtual Private Network

1. INTRODUCTION

In the *Software as a Service* (SaaS) service model, software vendors host software applications in the cloud in order to make them accessible over the internet. It is a cost-effective alternative to the traditional *Application Service Providing* (ASP) service model, in which the application is installed on the customer organization's premises and the customer must take responsibility of maintaining its own infrastructure in order to keep the application running. Having to maintain only a single scalable multi-tenant production environment is also beneficial for software vendors. According to Gartner, the increasing adoption of cloud-first strategies has brought the worldwide SaaS revenue to its all-time high, to 94.8 billion U.S. dollars in 2019, and the number will rise to 110.5 billion U.S dollars by 2020 [11].

Still, over the last decades, many legacy systems that have evolved organically into majestic ecosystems are still located on-premises, meaning that the system is operated and runs on servers on the premises of the organization using the software, rather than at a remote facility such as a server farm or cloud. According to Moyle, a legacy application is a technology that is difficult to replace, and which would be developed with different technologies today [59]. These systems are very high value for the business, which is the reason why they are kept and maintained by organizations [59].

This thesis was conducted as a research project for a Finnish IT service provider and software vendor (referred throughout this paper as *the company*). The company has faced the aforementioned challenge with a handful of mature legacy systems. On one hand, these legacy systems provide high value, because they have been well adapted to the existing business processes over the years, they maintain organizational knowledge and therefore provide significant competitive advantage [19], but on the other hand these systems are developed with technologies that are no longer widely mastered, and therefore there are less knowledgeable professionals available. Their deployment models often do not scale well due to including manual stages and customer environment specific quirks. Despite these challenges, the world keeps revolving and business requirements keep evolving, which poses a great need to keep expanding these systems and to add more features. Not unlike other software service providers, the company of-

fers many of these applications *off the shelf* (OTS/ASP), meaning that they are not entirely tailored for each customer's specific needs, but instead each system is offered and provided to several customers with *instances* of the same system running on each customer's premises, i.e. in each customer's *environment*, with customer specific *configurations*. This manual delivery to each customer's on-premises environments is portrayed in Figure 1.

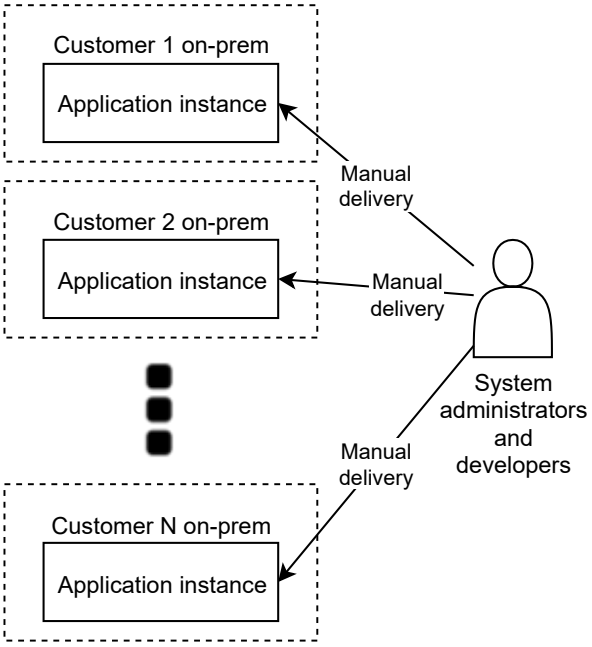

Figure 1. *The OTS/ASP deployment model. Delivering new application versions includes manual steps performed by system administrators and developers.*

Cloud migration is the process of partially or completely moving digital assets, services, IT resources or applications to a cloud platform with an intent to improve certain quality attributes, such as scalability, maintainability and performance [61]. Given the long-term benefits of cloud migration, many organizations and IT service providers decide to do so. Cloud migration does not come without risks. Especially with legacy applications it may require costly rewrites and be time-consuming [28]. In addition to the challenges, mature legacy systems also have perceived benefits, such as reliability and stability, which is why many businesses still host their most mission-critical applications on-premises, such as *Enterprise Resource Planning* (ERP), *Customer Relationship Management* (CRM) and payroll [23].

The challenges involved with a fully-fledged cloud migration can be mitigated by using a *hybrid cloud* deployment model [48]. In a hybrid cloud deployment, some system components are in the public cloud and the rest on-premises. This implies that there must be a mean of communication over the internet between these two environments. In the context of the company, this would mean that the monolithic legacy systems (i.e. *master*

systems) and their existing ecosystems could potentially be left untouched, and any new functionality could be developed as *add-on applications* with modern technologies and deployed to the public cloud. Because all the master systems are OTS/ASP, i.e. they are deployed to several customer environments, the new add-on applications could leverage a *multi-tenant* approach, making it necessary to have only a single multi-tenant instance of each add-on application running in the public cloud, shared by multiple customers. Figure 2 portrays this basic idea of the *Multi-Tenant Hybrid Cloud* (MTHC) architecture:

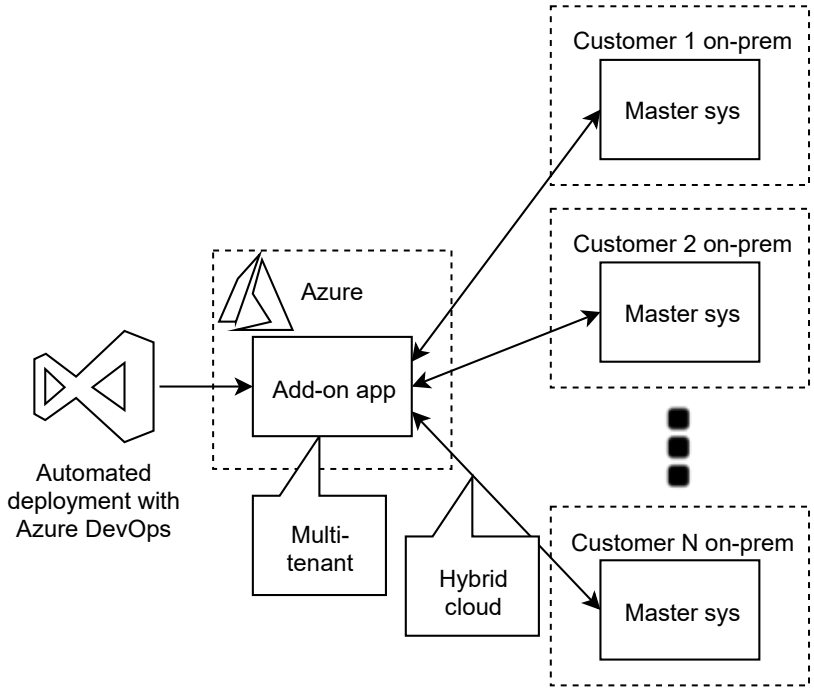

Figure 2. *The MTHC architecture. A multi-tenant add-on application, deployed with Azure DevOps Pipelines and running in Azure, must have hybrid cloud connectivity to on-premises resources owned by the master system.*

In this thesis the MTHC architecture is considered in two new add-on application projects. The add-on applications are:

- **Case 1: *Data Quality Management* (DQM) application** with *Dealership Management System* (DMS) as the master system
- **Case 2: *Tire Hotel Personal Digital Assistant* (THPDA) application** with *Tire Hotel System* (THS) as the master system.

The specifics of these applications are discussed in chapters 7 and 8 respectively. Certain goals were set for the MTHC architecture. Reaching these goals is assessed in the evaluation section of each of the cases. The goals are:

- **Goal 1: Develop the add-on applications with modern technologies:** Allow developers to use technologies that are widely supported and that they are already familiar with. This makes it easier to incorporate new developers to these software projects.
- **Goal 2: Make delivery automated:** In order to make the delivery process scalable and manageable despite the number of customer organizations using each add-on application, i.e. *subscribed* to each application, the application delivery processes must be automated.
- **Goal 3: Make subscription automated:** Including new customers should not require complex manual operations. Ideally it should be only a question of connectivity between the add-on application and the master system.
- **Goal 4: Share cloud resources between all customer organizations:** The deployment model of the add-on applications should support sharing their allocated cloud resources between all customer organizations, providing cost savings and decreasing the applications' cost-per-customer as the number of subscribing customer organizations increases.

In chapters 2, 3 and 4, the theoretical concepts behind cloud computing and MTHC are discussed and the essential concepts related to the MTHC design are introduced. Chapter 5 explores the MTHC design challenges with a literature review. The aim of this literature review is to gain an understanding of the challenges, which would be beneficial considering the planning and implementation of the development projects. In chapter 6 MTHC enabling technologies are discussed. Due to organizational reasons, the cloud technologies are limited to *Microsoft Azure* offering and the add-on application development stack will be *ReactJS/ASP.NET Core*. Git version control and the CI/CD pipelines are provided by Azure DevOps. Chapters 7 and 8 present the two software development projects from the point of view of the MTHC architecture. In chapter 9 the study is summarized and evaluated.

2. CLOUD SERVICE MODELS

Something as a Service is a category of *service models* in which a *Cloud Service Provider* (CSP), such as Microsoft Azure, or a software vendor delivers scalable cloud-based services on-demand over the internet [31]. From the CSP's point of view, *something as a service* utilizes resource pooling to serve multiple customers over the same infrastructure, and should be well measured in order to enable quick on-demand scalability and to provide transparency over the resource usage to its customers [31]. *Something as a Service* recognizes a large amount of different technologies, tools and products. The *National Institute of Standards and Technology* (NIST), an organization that develops standards and guidelines, recognizes three service models: *Software as a Service* (SaaS), *Platform as a Service* (PaaS) and *Infrastructure as a Service* (IaaS).

In SaaS, customers are provided the capability to use software vendor's applications that run in a cloud environment [31]. The applications are accessible from various client devices, typically through a web browser or a local program interface [31] (e.g. Microsoft Outlook [55]). There is a great deal of SaaS applications available for personal use for free and for organizational use as paid services, including many sophisticated business applications such as CRM and ERP applications [55]. The consumer organization does not manage or control the underlying cloud infrastructure, such as network, servers, operating systems or storage [31]. In many cases it is beneficial for organizations to have their applications client-free and data stored in the cloud, available from anywhere [55].

According to NIST, PaaS allows the consumer to deploy consumer-created or acquired applications – created using programming languages, libraries, services, and tools supported by the CSP – onto the cloud infrastructure [31]. This means that PaaS permits developers to create and deploy applications using built-in software components provided by the CSP, with many cloud features such as scalability, high-availability and multi-tenant capability included within the platform, reducing the amount of code that developers must write [54]. In PaaS, the level of *service abstraction* is lower than in SaaS, because – despite not having to manage or control the infrastructure – the consumer has control over the deployed applications and possibly configuration settings for the hosting environment [31]. As an example, Azure App Service [40] and Heroku are PaaS offerings.

NIST describes IaaS as a capability that allows the consumer to provision processing, storage, networks and other necessary computing resources [31]. The consumer can

run arbitrary software on top of the provisioned infrastructure, including operating systems and applications [31]. The consumer does not manage CSP's physical cloud infrastructure, but has control over the provisioned virtualized infrastructure, such as virtual machines, networks, storage and deployed applications [31]. Being able to provision virtualized infrastructure easily can make it less expensive and less complex to manage and deploy applications, such high-performance computing, big data analysis and plain web applications, compared to a situation where the consumer has to manage physical hardware by itself [53]. This also makes it easier to innovate rapidly and to set up and dismantle test and development environments, bringing new applications to market faster [53].

The NIST definitions were published in 2011 and the cloud paradigm has evolved quite a bit ever since. Especially *serverless* or *Function as a Service* (FaaS) has become an established service model since then. FaaS (e.g. Azure Functions, AWS Lambda) allows executing a small piece of software, *a function*, without requiring the software service provider to manage servers or complex operational aspects [57].

The NIST model's validity has been challenged by, for example, Miyachi [57]. Instead of differentiating service models based on the service abstraction, Miyachi's model differentiates them based on the applications they are suitable to be used for and by the end user they are used by. These Miyachi's service models are presented in Table 1.

Table 1. *Miyachi's service models [57].*

Service model	**Applications**	**End user**
App Services	Cloud apps, Social media apps	Any user
Built-up PaaS	Business as a Process, Data Analytics	Rapid developers
Serverless Computing		Speed developers
PaaS		Developers
Foundational PaaS	Containers, Messaging, Object Storage	DevOps
Software Defined	Virtual Machines, Software Defined Networks	Infrastructure engineers
Hardware	Services, Switches, Routers, Storage	

Miyachi's model does not include IaaS per se, but instead has separated it into three levels of virtualization, *Hardware* level being the least and *Foundational PaaS* being the most virtualized levels. Miyachi's model differentiates traditional *Software Defined* service models (e.g. virtual machines) from the more recent, cloud native *Foundational PaaS* based delivery models (e.g. application containers). Due to being focused on deployment of binaries, *Foundational PaaS* can be viewed as a traditional platform as a-service for *Development Operations* (DevOps*)* purposes, whereas Miyachi's *PaaS* and *Built-up PaaS* are platforms for coders (focus is on the deployment of code) and for

power users (focus is on the deployment of higher-level business models) respectively. Due to abstracting out most of the infrastructural concerns of *PaaS*, Miyachi positions *Serverless Computing* between *PaaS* and *Built-up PaaS*. The highest level of abstraction is *App Services*, which includes applications that are meant to be used by business users without technical know-how.

3. MULTI-TENANCY

Software vendors often adopt SaaS as their primary service model because SaaS allows them to offer their software services over the network, reaching a global market. However, in order to gain the cost efficiency of a shared infrastructure, the offered software applications must be *multi-tenant* [73].

According to the systematic mapping study carried out by Kabbedjik *et al.*, "multi-tenancy is a property of a system where multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant" [27]. The term "transparency" in this definition implies that the end-user should not be required to be aware of any multi-tenancy aspects of the system, but instead the user experience should be equivalent to the one of a dedicated system, e.g. OTS/ASP.

In their paper, Bezemer & Zaidman claim that the definitions for multi-tenancy found in literature are vague and therefore they define the concept by themselves as follows: "A multi-tenant application lets customers (tenants) share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it ran on a dedicated environment" [3]. According to Bezemer & Zaidman, a "tenant" is both an organizational entity which rents a multi-tenant SaaS solution and a group of users that are the stakeholders in the organization [3].

Chong & Carraro define multi-tenancy through an example: when a user from one organization uses a multi-tenant CRM application service to access her employer's customer data, the application instance that the user connects to may be accommodating users from dozens, or even hundreds, of other organizations, all completely oblivious to each other [8]. This requires an architecture that maximizes the sharing of resources across tenants, but that is still able to differentiate data belonging to different customer organizations. In their paper focusing on multi-tenant variability, Walraven *et al.* refer to this definition of multi-tenancy [72].

The systematic literature mapping of Kabbedjik *et al.* provides a definition of multi-tenancy that aggregates several distinct literature sources, but they still remark that there may be research bias in their results [27]. Their study also includes both Bezemer & Zaidman's and Chong & Carraro's definitions. Bezemer & Zaidman formulated their own

definition for needs of their study, but this was mainly due to their study focusing on concrete caveats of multi-tenant architectures, and they claim that the abstract definitions found in literature were too vague for their purposes [3].

According to Walraven *et al.*, the main benefit of the multi-tenant approach is that the system's operational costs can be significantly reduced by using hardware and software resources more efficiently, multiplexed across customers, and by simplifying the overall maintenance effort [72]. Maintenance is simpler and more scalable, because upgrading the software can be performed for all tenants at once [72]. On the other hand, there are several challenges, such as having to manage an expanding set of variations in the software implementation as well as in configurations. *Variability* is the extent that a software component functionality can be manipulated with tenant-specific customization and configuration. Components with no variability always produce the same outcome for all tenants. The challenges are elaborated further in the literature review in chapter 5.

According to Kabbedjik *et al.*, the difference between multi-instance (i.e. *instance per tenant*) systems (e.g. traditional OTS/ASP) and multi-tenant systems is that for multi-instance systems it is not by definition necessary to have shared resources, because a new system instance can be deployed for each customer separately [27]. On the other hand, the key difference between multi-user and multi-tenant systems is that in multi-user systems the same invariable functionality is provided for all customers, whereas in multi-tenant systems the application functionality may be differ between users of different tenants and, on the contrary, be alike between users of a single tenant [27]. A software vendor may, for example, customize its multi-tenant application offering to suit the needs of one of its customers without the other customers experiencing any difference.

Walraven *et al.* elaborate the division between multi-instance and multi-tenant systems even further, dividing multi-tenancy to three *Levels of Tenancy* (LoT): *shared infrastructure, shared middleware* and *shared application* [72]. In addition to these three, this thesis considers the lack of multi-tenancy, i.e. having an *instance per tenant*, as the lowest level of tenancy. The comparison between these four is shown in Figure 3.

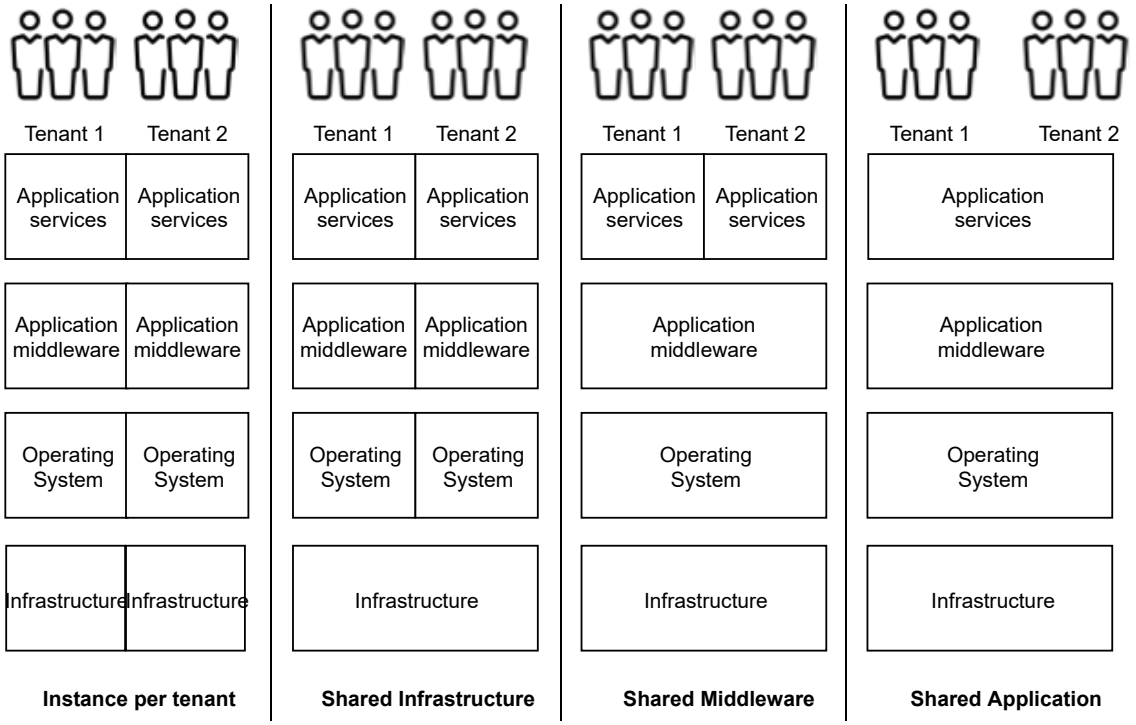

Figure 3. *The levels of tenancy.*

In the *shared infrastructure* approach, as the name says, only the infrastructure (e.g. virtual machines) is shared between the application processes dedicated to different tenants. On the contrary, if the *shared middleware* approach is used, the application processes are shared between tenants, and thus also the operating system, but *application services* are not. Walraven *et al.* do not describe in detail any operational aspects of a "middleware", but in the context of this paper it is reasonable to consider a middleware as a web application request middleware, which will be explained in detail in subchapter 6.1.2. With the *shared middleware* approach, the initial engineering complexity is shifted from the application service level to a reusable middleware layer [72]. This implies separate application service instances for each tenant and therefore higher maintenance costs [72]. In their paper, Walraven *et al.* propose the *shared application* approach for building true multi-tenant applications with the flexibility to support tenant-specific requirements [72]. Since all tenants are served by the same application services, the services must be designed to support tenant-specific variability [72].

The *shared application* model proposed by Walraven *et al.* is essentially an architectural pattern consisting of a specific *multi-tenancy support layer* that enables tenant variability. Since this model is suitable for web applications and because the model introduces several overarching concepts related to multi-tenant application design, the model is briefly discussed in this subchapter. The model is presented in Figure 4. The very model presented in the figure substitutes the concept of *Feature*, used by Walraven *et al.*, with

Service, because this is more in-line with the later implementation stages regarding this thesis.

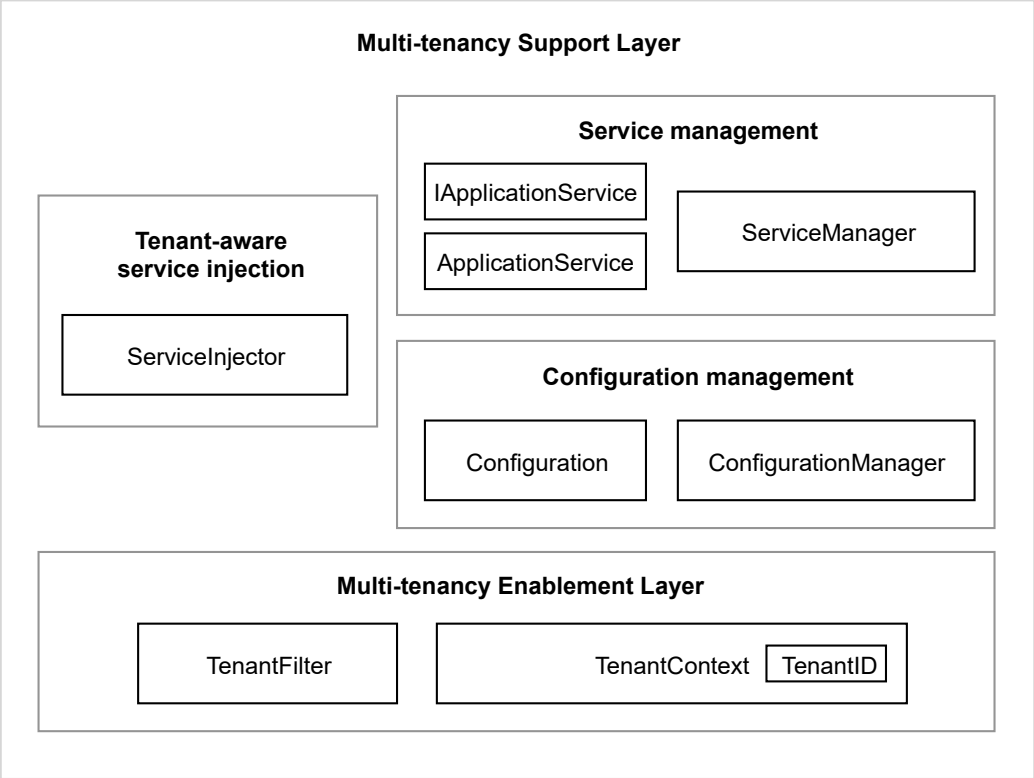

Figure 4. *Overview of the multi-tenancy support layer [72].*

The *Multi-Tenancy Enablement Layer* (MTEL) consists of two components: a *TenantFilter* and a *TenantContext*. For each incoming *Hypertext Transfer Protocol* (HTTP) request, the MTEL resolves the tenant linked to the request. Typically tenant specific requests contain some tenant identifier, a *TenantID*, that the MTEL tries to extract. Once the requesting tenant is resolved, a *TenantContext* is created. [72]

The current *TenantContext* bears the resolved *TenantID*. *TenantContext* is available for all multi-tenant *ApplicationServices*, i.e. application service implementations, and thus the actual functionality of each *ApplicationService* can vary between different tenants (linked to different *TenantIDs*), depending on how each tenant is *configured*. In some cases, the application can depend on an abstract *IApplicationService* instead of a specific implementation. *Customized ApplicationService* implementations can be bound to each *IApplicationService* per-tenant basis. [72]

ApplicationServices and the *TenantContext* use a tenant specific *Configuration* as the source of *configuration settings* for the tenant [72]. A *Configuration* instance is resolved with the *ConfigurationManager* using the current *TenantID*. Each tenant has a single dedicated *Configuration* instance. An example of a configuration setting would be any

tenant specific user interface theming or a tenant specific connection string to a tenant specific database. The difference between configuration and customization is discussed in subchapter 5.5.2. The *TenantFilter* is used for filtering multi-tenant database queries and commands [72], so that, for example, only the *Structured Query Language* (SQL) database table rows owned by the requesting tenant are considered during a query execution.

Walraven *et al.* explain that in their proposed model the current the current *TenantContext* can be made available to *ApplicationServices* using *Dependency Injection* (DI). Instead of instantiating *ApplicationServices* directly inside other *ApplicationServices*, the flow of control is inverted: the life cycle management of *ApplicationServices* is controlled by a DI container, the *ServiceInjector* [72]. *ServiceInjector* takes care of binding each *IApplicationService* to a concrete *ApplicationService* for each tenant and providing them the correct *Configuration* instance [72]. DI in ASP.NET Core context is discussed further in chapter 6.

4. HYBRID CLOUD

According to NIST, there are four *cloud deployment models*: *private cloud, community cloud, public cloud* and *hybrid cloud*. In a private cloud deployment, the cloud infrastructure is provisioned exclusively for a single organization [31]. This cloud deployment can be owned, managed and operated either by the organization itself or by a third party, and it may exist *on-premises* (or just on-prem) or *off-premises* [31]. Often “on-premises” is used as an opposite of SaaS, meaning that if something is performed on-premises, it is performed within an internal corporate network, disconnected from any cloud-based SaaS services. Private clouds are often used by mid- to large-size organizations with business-critical operations, seeking enhanced control and security over their environment [48].

NIST defines community cloud as a cloud infrastructure that is provisioned for exclusive use by a specific community of consumers from organizations with common concerns, such as requirements for security or policy compliance [31]. Like a private cloud infrastructure, a community cloud infrastructure can be managed by the community organizations themselves or by a third-party service provider, and it may reside on- or off-premises [31].

Public cloud is the most common deployment model for cloud computing [48]. According to NIST, a public cloud infrastructure is provisioned for open use by the general public, and it is owned, managed and operated by a CSP (i.e. not the consuming organization) [31]. Public clouds are multi-tenant services, meaning that multiple organizations (tenants) share the same physical infrastructure, such as servers and network devices [48]. As opposed to private cloud, costs are generally lower with a public cloud infrastructure, because hardware is not necessary to be purchased and maintained by the tenants [48]. Public clouds offer near-unlimited scalability, because resources like computing capacity are available on-demand [48].

NIST defines hybrid cloud as a cloud infrastructure that is a composition of two or more distinct (cloud) infrastructures that remain unique entities, but are *bound together* by a standardized or a proprietary technology that enables data and application portability between the environments [31]. By combining on-premises resources with public clouds, organizations can reap the advantages of both. One can, for instance, use public cloud for high-volume, lower-security needs (such as web-based email) and the private cloud/on-premises environment for sensitive, business-critical operations like financial

reporting [48]. In addition to this *vertical positioning*, which means that the services located in the public cloud and on-premises are not the same, *horizontal positioning* is also used in some scenarios. Horizontal positioning means that the services located in different environments are equal. One motivation for using horizontal positioning is *cloud bursting* in which a service runs on-premises until there is a spike in demand (such as a seasonal event like online shopping), at which point the services can “burst through” to the public cloud to tap into extra computing resources [48].

5. LITERATURE REVIEW

The literature review was conducted using the *Systematic Mapping Study* (SMS) process described by Petersen *et al.* [63] as the basis for the process. The difference between the process described by Petersen *et al.* and the exact process used in this literature review is that the goal of this literature review was to gather a set of challenges related to the MTHC architecture. For this reason, the final mapping step was replaced with reporting of the classification results using the source literature. Figure 5 illustrates the study process flow.

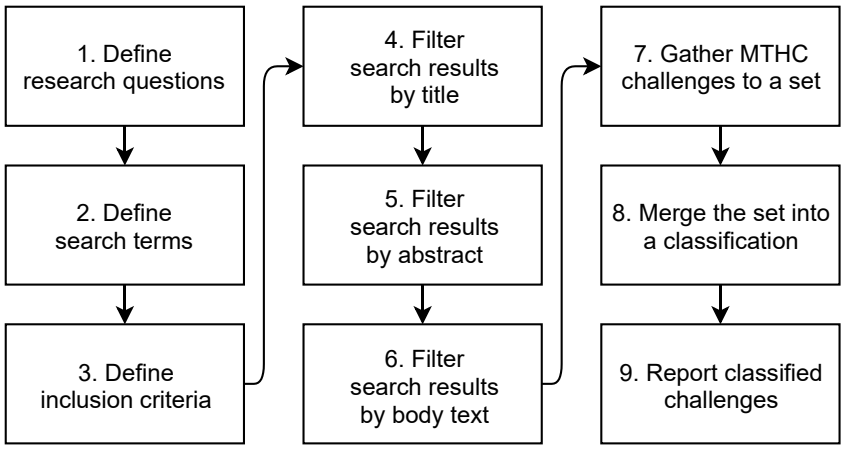

Figure 5. *The literature review process.*

Petersen *et al.* point out that SMS is suitable for gaining a coarse-grained overview of a research area [63]. Lacking enough knowledge about MTHC design challenges, the main goal for the literature review was to form a general understanding about this subject. Hence, SMS was a suitable choice as the basis for the process model. The subchapters of this chapter walk through the steps of the literature review process and describe how they were carried out.

5.1 Definition of research questions

The ultimate goal of the literature review was to form a more thorough and applicable understanding on the caveats of the MTHC architecture. For example, what information security challenges does a multi-tenant hybrid cloud design pose? This “challenge” standpoint was used in forming the research question:

Research question: *What software design challenges should be considered particularly when a vertical hybrid cloud deployment model is used for accessing on-premise resources in a multi-tenant application?*

The answer to the research question would preferably be an elaborate classification of non-functional considerations.

5.2 Definition of search terms

The search was conducted using Scopus as the bibliographical source. Initially, the search focused specifically on the MTHC architecture. Unexpectedly, this led to very little and mostly irrelevant results. Ultimately, the search was conducted in two parts: one search for multi-tenancy and one for hybrid cloud. This approach worked sufficiently well, because the two concepts are very independent.

Table 2 describes the search terms that were used for each concept. It also displays the numbers of search results and papers after the filtering was performed.

Table 2. *Search terms.*

Concept	Search term	# of search results	# of papers after filtering
Multi-tenancy	(“multi-tenan*” OR “multitenant*”) AND (“per tenant” OR “tenant?specific” OR “each tenant” OR “all tenants”) AND “cloud” AND (LIMIT-TO(SUBJAREA, “COMP“))	28	17
Hybrid cloud	TITLE-ABS-KEY(“hybrid cloud” AND (“on?premise*” OR “internal” OR “intra” OR “enterprise”))	191	15

The focus within multi-tenancy literature varies a lot. Some papers give multi-tenancy a very brief honorary mention, whereas some give major attention to very specific aspects of multi-tenancy. The set of search results was possible to be refined by including search terms that are typical for literature that focuses specifically on multi-tenancy (and not just briefly mention the concept). This excluded lots of false positives.

Considering hybrid cloud, the result set was not possible to be narrowed down as much, so the final set of search results contained more false positives. The search terms attempt to emphasize specifically vertical hybrid cloud positioning.

5.3 Definition of inclusion criteria & filtering

Filtering was done in three phases, each phase reducing the size of the set of papers. First, the initial set of papers was evaluated by their titles. Then the resulting set was evaluated by abstracts and finally this set was evaluated by body texts. In each phase the inclusion criteria were the following:

Title/abstract/paper addresses general design patterns or architectural aspects of hybrid cloud or multitenancy

AND is available via Scopus

AND considering multi-tenancy literature

- is not focused around a specific technology offering.

AND considering hybrid cloud literature

- is not focused solely on cloud bursting, offloading, high availability or achieving better quality of service with a hybrid cloud deployment
- is not focused just generally on cloud computing
- is not focused solely on hybrid storage
- involves the perspective of software service provider (instead of CSP)
- is not focused on hybrid IaaS or cloud desktops.

The filtering process ended up with 32 papers: 17 papers for multi-tenancy and 15 papers for hybrid cloud. Figure 6 displays the publish years of the chosen 32 papers.

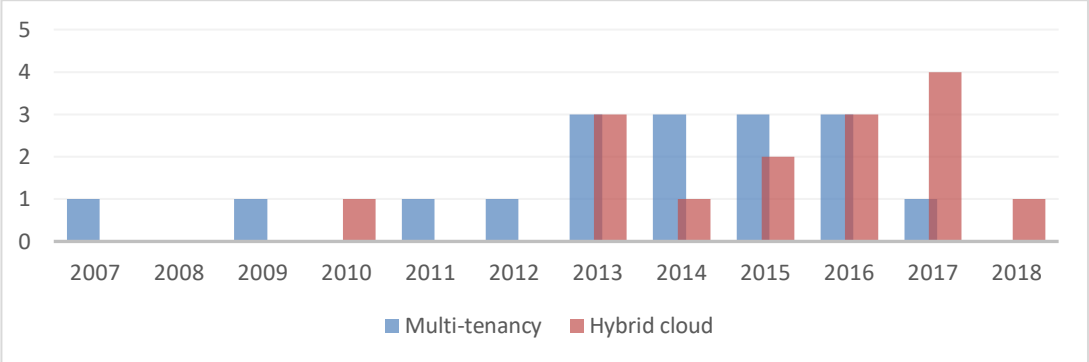

Figure 6. *Paper publish years.*

As shown in the figure, both of the concepts have been around for about a decade. The publication frequency has sustained quite stable over time. This is not a surprise, because multi-tenancy is a very typical pattern in cloud SaaS applications, and on the other hand, hybrid cloud is a typical deployment model for cloud migration.

5.4 Gathering & merging the MTHC challenges

According to Petersen *et al.*, keywording is done in two steps: first the reviewers read the abstracts and look for keywords and concepts that reflect the contribution of the paper [63]. When this is done, the sets of keywords are combined together in order to develop a high-level understanding about the nature and contribution of the research [63]. The final set of keywords can be clustered and used to form the categories for the systematic map [63]. In this literature review, keywording was performed during step 7 (see Figure 5). Then this relatively duplicated set of miscellaneous keywords was cleaned by merging overlapping keywords together. This process resulted with a classification of the MTHC architecture design challenges shown in Table 3.

Table 3. *The classification of MTHC architecture design challenges.*

Multi-tenancy

- *Variability*
  a. Over the application architecture
    i. Infrastructure variability
    ii. Application level variability
    iii. Persistence variability
  b. Over the application lifecycle
  c. Self-service variability
- *Service Level Agreement (SLA)*
  a. Tenant interference
  b. SLA variability
- *Security*
  a. Tenant isolation
  b. Compliance to regulations

Hybrid cloud

- *Security*
  a. Network security
  b. Authorization
  c. Compliance to regulations
  d. Governance
- *Partitioning*
  a. Data
  b. Application
- *Connectivity*
  a. Connectivity technology
  b. Performance

One downside of this classification, compared to results of typical systematic mapping studies, is that this classification does not take into consideration the *importance* of each challenge. Typical systematic mapping studies report the number of papers focusing on each entry in the classification. In case of this literature review, formulating these statistics would have been difficult, because, as will be discussed in subchapters 5.5 and 5.6, the final classification is quite interdependent, meaning that certain challenges occur together or affect each other in some ways. From the point of view of the business goal of the literature review, i.e. forming a set of design challenges to assist the actual design and implementation efforts in the oncoming software projects, this is not an issue.

5.5 Multi-tenancy design challenges

This subchapter reports the classified set of design challenges related to multi-tenancy. The quality of multi-tenancy literature varies quite a lot. Some papers focus on very specific problems, whereas some papers, such as the one produced by Walraven *et al.* [72][73], provides more exhaustive perspectives on the challenges related to multi-tenant design. For this reason, the papers by Walraven *et al.* are referred to quite a bit throughout this subchapter.

5.5.1 Infrastructure variability

According to Walraven *et al.*, tenants may demand that the underlying application processes of their SaaS subscriptions should be executed in specific environments [73]. For example, some tenants may require a lower level of latency or ability to decide the location of their data storage. Therefore, not only application's functionality may vary among tenants, but also the cloud infrastructure itself is subject to variability. Horcas *et al.* add

that making the application independent of an exact infrastructure decreases SaaS vendor lock-in from the customer's point of view [22].

Cloud infrastructure variability is an essential factor also in the context of hybrid cloud applications, as will be discussed in subchapter 5.6. For example, it may be crucial for some tenants to be able to behold their business-critical data on-premises, whereas it may not be as critical for others.

5.5.2 Application level variability

In the literature, two kinds of application level variability approaches were discussed: build-time static variability by *customizing* the implementation; and runtime dynamic variability with *configuration*.

Customization (or tailoring), involving code modifications and an inevitable application deployment, is a more traditional approach for enabling application level variability [5][10][58]. Customization is used for satisfying tenant-specific requirements that cannot be achieved with the one-size-fits all core functionality of a SaaS application [72][73].

Configuration, on the other hand, supports application level variability by parameterization [10][73] that affects the application execution during runtime. It can be performed with e.g. configuration files, that the application reads during runtime, environment variables or command line parameters. According to Walraven *et al.*, compared to customization, configuration is generally the less expensive approach for service providers because tailored implementations introduce a new layer of complexity and additional maintenance overhead [72]. It should be noted that enabling and maintaining a *configurability model* in a multi-tenant application may be a non-trivial issue as well, as is later discussed in this chapter.

When designing the strategy for application level variability, the LoT of the application must be considered. Correia *et al.* claim that the traditional ASP models of software deployment benefit from customization, because separate builds that include their dedicated customizations can be deployed for each individual customer separately [10]. As discussed, this approach uses the *instance per tenant* LoT. On the other hand, if the *shared application* LoT is used, implementing customizations for one tenant comes with a risk of having unintentional side-effects for other tenants [5]. They may be manifested as, for example, downtime during deployment, performance interference, fault propagation or as new bugs in existing shared functionality. Hence, service providers should make sure that any tenant specific customizations are not applied at the expense of *tenant isolation* [73].

There are major trade-offs between customization and configuration. On one hand, with customization, a specific customer requirement can be satisfied without a sophisticated configurability model [73]. On the other hand, it implies adding customer-tailored implementation to the code base which has to be maintained [70][72]. A configurability model can easily become overly complex and make the application difficult to maintain [58][70] and increasingly difficult to be validated as the model evolves [17][70]. Variability model evolution is further discussed in subchapter 5.5.4.

In the literature, several techniques are proposed for enabling application level variability. Considering the *instance per tenant* approach, *Software Product Line* (SPL) techniques are proposed for customization management [10][58][73]. According to Babar *et al.*, a SPL is a set of software-intensive systems that share a common set of core features for satisfying the needs of a particular market segment, making it possible to reduce software development cost by reusing code assets [2]. In the context of multi-tenant applications, a SPL would be beneficial for satisfying the specific needs of a particular tenant instead of a particular market segment. It should be noted that with SPL the burden of ensuring tenant isolation is not eliminated but shifted from application *runtime* to *build-time*: Applications should be built in such a way that the tenant specific customizations are included within tenant specific builds and deployments. In addition to this build-time isolation, several ways of implementing SPLs are proposed in the literature. Object-oriented programming patterns discussed in the literature include *reflection*, *aspect-oriented programming* and *dependency injection* [73].

If a SaaS application uses the *shared application* approach, it has to include a MTEL [5][72][73]. In web applications this means request time instantiation of tenant specific objects, parametrized with tenant specific configuration, as was discussed in chapter 3.

5.5.3 Persistence variability

According to Correia *et al.*, tenant specific *data model extensions* are a form of multi-tenant application customization, including new *entity attributes* (e.g. new table columns) and new *entity types* (e.g. new tables) [10]. Data model extensions must be combined with any necessary application level customizations or, possibly, with the application's configurability model that integrates the additional attributes to the functionality of the application [10]. Data models are extended due to some tenant specific functionality requiring attributes that are not present in the one-size-fits-all core functionality of a SaaS application [25]. *Relational Databases* (RDB) are typically too strict to enable this [25]. According to Foping *et al.*, this is due to the concept of multi-tenancy not being a first-class citizen in any well-known persistence technology offering [17]. Support for multi-

tenant persistence and variability must be implemented case by case or by utilizing a third-party application level variability middleware offering.

In the literature, similar LoT approaches are considered for persistence as were proposed for application instances [17][25][68][74][75]:

- *separate databases (or shared machine)*
- *shared databases (or separate schemas, shared process, table prefix)*
- *shared tables (or shared schemas).*

In each of the three approaches, tenant specific data is organized differently, so they vary in terms of tenant isolation. In a database modeled according to the *shared tables approach*, tenant isolation is the lowest, because each tenant's data is stored in tables that are shared between all tenants [68]. Data ownership is distinguished by a column containing the *TenantID*, which is interpreted and used by the application logic [68], i.e. the *TenantFilter*.

Persistence variability goals can be achieved in several ways. In the literature, three approaches were discussed: *Entity-Attribute-Value-model (EAV-model)*, *extension tables* and *custom columns*. The literature focused mainly on RDBs, but there is no reason why these models could not be applied with other data storage paradigms too.

EAV-model [25] (or *pivot tables* [73], *name-value pairs* [10]) extends the core data model of a SaaS application with a key-value structure. An EAV-extension table is joinable to relational tables and still avoids any static modeling (customization) of tenant specific entity attributes and types. Downsides of this approach are not widely discussed in the literature, but naturally, it would not be able to benefit as much from the optimization methods provided by traditional *Relational Database Management Systems* (RDBMS) such as indexing. If the underlying type system was desired to be used, the EAV-model approach would also not organize the data according to the business domain but by data types (i.e. one EAV-extension table for integers, one for timestamps and one for strings). Figure 7 portrays an example EAV-model. In the example *Vehicle* table is extended with *EavExtension* table. *EavExtension* entries are joinable to *Vehicle* entries by *VehicleId*.

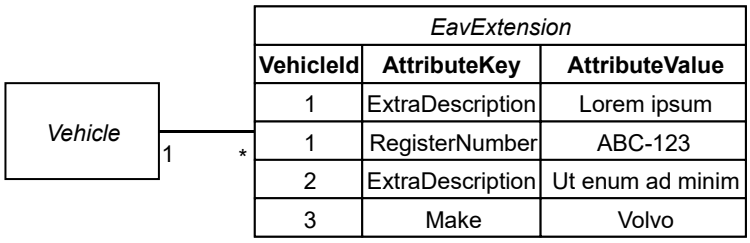

Figure 7. *EAV-model for persistence variability. The EavExtension table allows extending the static model of the Vehicle table.*

An *extension table* is a customized table created for the specific needs of a single tenant [25][74][75]. Extension tables are relational tables that refer to the tables that are part of the core SaaS functionality, and therefore can benefit from the RDBMS's optimization features. On the downside, this technique is less dynamic than the EAV-technique: extending the data model requires static schema modifications.

In the *custom column* and *sparse table* approaches core tables are extended with tenant-specific custom columns [5][10][17]. On the downside, if this approach is used in a *shared tables* data model, it may lead to tenant specific columns being shared by all tenants, decreasing the level of tenant isolation or at least shifting the responsibility to the application logic.

According to Walraven *et al.*, one of the key requirements for a multi-tenant execution platform is that all tenant-specific variations should be applied in an isolated way, without affecting the service that is delivered to other tenants [73]. Tenants should not be able to experience each other's customizations, such as *custom columns*, or access each other's data. The current *TenantContext* bears and provides the current *TenantID* that can be used by *data access objects,* implementing the *TenantFilter,* for filtering data that is stored in shared tables [73]. A data access object is an object for accessing data in a data storage, and it can be implemented with several software development patterns, such as *object-relational mappers* [74]. As an example, *EF Core 2.0* is an object-relational mapper that implements the *TenantFilter* with its *Global Query Filters*: A multi-tenant entity type should define an attribute for the *TenantID* and Global Query Filters ensure that within the current *TenantContext* only the data entries owned by the tenant can be queried and modified [43].

As mentioned, the literature does not explore the possibilities of more dynamic persistence techniques, such as document stores. Jastrow & Preuss claim that NoSQL (*Not Only SQL*), lacking many of the benefits of a RDBMS, would offer sufficient extensibility [25], but they do not dive into the details of this issue. For example, *Cosmos DB*, which is a multi-paradigm cloud native NoSQL storage offering in Microsoft Azure, supports multi-tenancy with its isolation-ensuring *partition keys* [37].

5.5.4 Variability over the lifecycle of the application

As the complexity of a multi-tenant SaaS application increases over time, it becomes gradually more difficult to refine without affecting the service continuity [22][71]. The evolution and versioning of its variability must be able to be maintained. According to Walraven *et al.*, in successful SaaS offerings, increasing amounts of variability is produced to service increasing amounts of tenants [73]. There are two key components that increase the difficulty of managing variability evolution in SaaS applications: the number of served tenants and the variability complexity.

This explosion of variability complexity also means that supporting and implementing functionality for new requirements is a challenge [73]. Supporting new features in a software application may require modifying the existing components, which may lead to unexpected, hard-to-debug effects on the existing features due to variability complexity [73]. Configuration and the configurability model itself must be able to be *validated* by the SaaS provider in order to ensure that modifications have not broken any existing configurations [73].

In the literature, especially SPL based approaches are proposed for resolving the issue of variability evolution [22][58][73]. According to Walraven *et al.*, a SPL would support maintaining co-existing tenant-specific configurations and facilitate the development and management of customizable multi-tenant SaaS applications without compromising scalability [73]. SPL seems to be a concept that, even in the recent years, is discussed in-depth in the literature, but there are few implementations and libraries out in the public. The SPL based approaches proposed in the literature each emphasize different challenges in variability management. For example, Horcas *et al.* point out that typical SPL based approaches suffer from not focusing on automating the evolution at the architectural level and from being obsessed on variability management in the scope of a single tenant [22]. This makes it more difficult to perform changes automatically and consistently, because it increases the complexity of changing the existing configurations for all tenants at the same time. In addition to SPL approaches, Walraven *et al.* briefly point out that enabling backwards compatibility in the internal and external interfaces of an application is a simple way to cope with evolution requirements in less complex scenarios [73]. Walraven *et al.* claim that, while being simpler than creating a SPL, ensuring backwards compatibility does not scale as well as the variability complexity increases [73].

5.5.5 Self-service variability

In some applications, it may be imperative to allow tenants' users (or at least super users or administrators) themselves to setup the configuration for their SaaS subscription

[5][10][25][72]. In these scenarios, the SaaS provider is responsible for providing customer support to the administrators [72]. Self-service configuration tasks are performed in tenant-isolated administrative user interfaces, e.g. dashboards or consoles [5]. Chang *et al.* point out that including self-service configurability is especially important if the SaaS application is offered for small-to-medium businesses [5]. This is because such businesses are generally readier to comply with one-size-fits-all models inherent to SaaS applications with high scalability and low customizability. SaaS providers, on the other hand, may offer customization as a paid service for those tenants who are less thrilled to conform to this model [5].

5.5.6 Performance degradation by tenant interference

Since the data and the workloads of multiple tenants coexist within the same infrastructure, multi-tenancy may lead to performance degradation experienced by one tenant caused by another [5][65][76]. Occasional peaks at workloads and user amounts are typical causes for performance degradation in shared execution environments [65].

As mentioned, delivering customizations for one tenant may manifest itself indirectly as downtime, new bugs, fault propagation or performance degradation for other tenants [5]. According to Chang *et al.* [5] and Yaish & Goyal [75], SaaS providers should put more effort in preventing this from happening. Chang *et al.* [5] and Su *et al.* [68] also claim that availability is one of the most important *Service Level Agreement* (SLA) metrics for hosted applications (such as SaaS), and therefore creates a challenge for multi-tenant systems. Downtime may lead to considerable financial losses for multiple customers [68]. This means that fault isolation (preventing faults propagating across tenant boundaries) is a key requirement for multi-tenant systems [5].

Fault isolation should be regarded from the point of view of the used LoT model: if the level of tenancy is low, faults have less opportunity to propagate over tenant boundaries. Despite being better for fault isolation, Walraven *et al.* point out that a low LoT has substantial disadvantages from the point of view of scalability [72].

The role of *scaling out* was not addressed in the literature. Despite that, having multiple instances of the same stateless service, multi-tenant or not, behind a load balancer does certainly increase the availability of the service and decreases the possibility of fault propagation. For example, if one process crashes due to a fault, the rest of them are still up and serving.

5.5.7 SLA variability

Not all tenants are equal: for some, disruptions are costlier than for others [71]. A SLA is by definition an agreement, which defines a quality compromise between the tenant and the SaaS provider [5][71]. Tenants may want to gain more control over the delivered service by imposing additional availability and performance requirements [68]. For this reason, the SLA itself can be a point of variability.

Little is discussed about how a SLA is transformed into a point of variability, but, for example, Walraven *et al.* propose that this is one of the challenges that can be resolved with SPLs [73]. A SaaS provider could naturally dedicate resources and, on the other hand, throttle each tenant according to the tenant-specific SLAs.

5.5.8 Tenant isolation

Multi-tenant SaaS applications should aim to isolate the data owned by one tenant from other tenants for preventing unauthorized access [5], because tenants do not trust each other by nature [5]. Due to the risks associated with unauthorized access, outsourcing to multi-tenant environments is still viewed risky by organizations [75]. In the literature, tenant isolation is seen as a concern of performance interference and data leakage due to shared data storage platforms [17][58][74][75] and shared computing platforms [5][75].

Requirement for tenant isolation can vary among tenants. For example, some tenants may approve storing their data in a shared storage, while others, who are more concerned about confidentiality, may pose a requirement for a physically separated storage in order to fully isolate themselves from other tenants [58][73]. Even further, in some scenarios some tenants may require an isolated computing environment [5]. According to Yaish & Goyal, it is unlikely that tenants would risk their business-critical data in favor of reducing the total cost of ownership with SaaS applications [75]. Chang *et al.* point out that isolation requirements are correlated with tenant sizes: a large enterprise may prefer to pay a premium for isolated application or storage instances to avoid any risks associated with resource sharing, whereas small-to-medium businesses may prefer investing in services with a reasonable quality at a lower cost, being less worried about the risks associated with the tenant isolation practice [5].

As was discussed earlier, there are three LoT approaches for persistence: *separate databases*, *shared databases* and *shared tables*. Chang *et al.* point out that the choice between shared and dedicated instances is always a compromise between scalability and tenant isolation (among other security concerns) [5]. In a *shared application*, since tenants desire to access and use the service as if they were the sole users [5], this com-

promise is generally dealt with *data level* and *application level security*. Data level security aims to secure the access to the data storage with methods such as data encryption by tenant specific encryption keys [5] and fine-grained authorization policies [75]. The *TenantFilter*, discussed in subchapter 5.5.3, ensures application level data security.

Comparing the two approaches, data level security is at least the more reliable approach for ensuring tenant isolation, because it relies on the authorization methods provided by the data storage itself (such as a RDBMS login and user access system). Application level security, on the other hand, can be more dynamic, more portable and easier to implement, as it is independent of and does not require utilizing data storage technology specific security techniques.

5.5.9 Compliance to regulations

Multi-tenant applications are constrained in many ways by legal and business-centric policies [30]. *General Data Protection Regulation* is one example of regulations that should be considered by both SaaS providers and customer organizations. It regulates the processing of personal and organizational data related to individuals in the European Union [13]. Isolation due to business-critical data processing may be extremely critical for some tenants, as was discussed in subchapter 5.5.1.

Compliance to regulations is also a point of variability. Tenants may have business policies and other drivers for integrating the SaaS application to their own internal monitoring or reporting systems [5]. Tenant specific authentication, especially, is a variability point of this kind [5][12][22]: it may be essential to allow tenants to authenticate using an identity authority of their choice. In practice, this may require creating integrations to the *Single Sign-On* (SSO) services, federating tenant user accounts to cloud identity management systems [5] or possibly leveraging a completely tenant-specific mechanism for user and tenant authorization (e.g. access policies) [12][22]. Decat *et al.* point out that supporting this is inherently complex, because the authorization should be able to be managed by different parties: SaaS provider should be able to manage authorization at the level of tenants and each tenant should be able to manage access at the level of users [12]. In addition, as mentioned, the authentication method itself may vary among tenants [12][22]. Not discussed in the literature, but many organizations have some SSO offering already in use, that can often be used as the *identity authority* (e.g. Azure Active Directory). An identity authority is a service that is able to authenticate identities either directly, e.g. check the username and password, or indirectly, e.g. issue *JSON web tokens* (JWT, explained in subchapter 6.1.2). Naturally, one should not assume that this is the case for all tenants.

5.6 Hybrid cloud design challenges

This subchapter reports the classified set of design challenges related to the hybrid cloud deployment model.

5.6.1 Network security

In a hybrid cloud deployment, the communication between the on-premises environment and the cloud resources must be secured, as the public network is used for data transmission [69]. For ensuring end-to-end security, enterprises must develop and manage secure interfaces with their SaaS providers [21][67]. Depending on the setup, SaaS providers may be able to provide this as a separate service to their customers [21]. Regardless of the specifics of the setup, SaaS providers must secure their endpoints and, as an example, ensure that incoming requests are authorized. Authorization will be further discussed in subchapter 5.6.2.

Customers must be able to comply with the client-side requirements of the SaaS provider. It is not unlikely for customers to be required to setup firewall rules for their internal environments, so that hybrid cloud applications may have access to their internal resources [7][62][69]. This issue is made more difficult to cope with by the fact that public cloud applications in a public network may have dynamic *Internet Protocol* (IP) addresses or use a range of IPs, complicating the management of firewall rules [62]. This is true for many inherently multi-tenant platforms, such as Azure App Service plan, which has several outbound IP addresses, depending on the physical server on which the process is currently being executed on [44]. In addition, according to Toosi & Buyya, assigning public IP addresses to all servers in a hybrid cloud SaaS application is not feasible in many cases and would be waste of resources [69]. Depending on the internal network setup of the enterprise, there may also be network address translation policies that prevent any attempted access to the internal resources from the public network [69].

In the literature, *Virtual Private Network* (VPN) is often proposed as a solution for the aforementioned networking challenges [6][67][69]. It resolves the challenges associated with the dynamic nature of the public network and makes it possible to deny public access to the network itself. Connectivity technologies are further discussed in subchapter 5.6.7.

5.6.2 Authorization

SaaS providers and customers rely on each other to provide accurate, correct requests: ones that can be *authenticated* (the requesting user or process can be identified) and

authorized (the requesting user or process is allowed to perform the attempted actions) [21]. The SaaS provider is responsible for implementing and providing interfaces with appropriate policies for authentication and authorization. In a SaaS application, each request will trigger several actions before the request itself can be fulfilled: the application must authenticate both the request (so that its origin is valid, i.e. the correct tenant) and the requestor. The SaaS application must be aware of the *user identities* of each tenant and therefore the user lifecycle management of each tenant must be extended to cover the requirements of the application [21], which is one of the key challenges in hybrid cloud integrations [20][66]. As was discussed for multi-tenant applications, cloud federation or the existing SSO services could be leveraged here for authentication.

5.6.3 Compliance to regulations

As discussed in chapter 4, one of the benefits of the hybrid cloud approach is the ability to keep sensitive data and sensitive operations on-premises. This facilitates protecting the privacy of data and to comply with requirements for data location and regulatory requirements [7][18][29][69]. On the other hand, a large amount of enterprise data may not be too business critical or sensitive for storing within and accessing from a hybrid cloud deployment [69].

One approach is to keep the master version of data on-premises and project a necessary subset to the cloud [29]. For instance, sensitive databases (e.g. related to credit card processing) could be located on-premises, while less sensitive components could be migrated to the cloud [4].

5.6.4 Governance

While initial fears of potential cloud adopters focused on the security of cloud environments in general, most analysis has now shifted its focus to governance aspects [21]. In organizations, there is still a lack of understanding of what changes when moving to the cloud and how to demonstrate compliance of these environments for regulatory purposes [21]. According to Hinton, customers who do not have a tradition of paying attention to security, or who believe that they have “good enough” security, may be unpleasantly surprised when their “good enough” for the internal environment is not good enough for cloud [21].

In a hybrid cloud deployment, organizations will essentially store data on platforms and locations which either organizations or users have little control over [66]. From organizational point of view, it is important to either have trust on the platform or to be able to verify the transitions and storage locations of its data [66]. Organization must decide

which data can safely be transferred to public cloud and to maintain accurate information about which data has been processed by which clouds [66][67]. Data asset value must be expressed clearly and in detail and possible risks and other side effects with third-party involvement must be evaluated [67]. Hinton adds that if an organization does not already have sophisticated governance practices in place, then migration to a hybrid cloud setup can be a great risk for them [21].

Generally, organizations want to have greater visibility to the platforms that are integrated with their on-premises environments for ensuring that their data and resources are not compromised [4]. One reason for this is the multi-tenant nature of many cloud environments [4] and, for example, the risk of data breach. For gaining better visibility, a hybrid cloud deployment could be integrated with existing organizational tooling [4], or the SaaS provider could include e.g. an easy-to-access security dashboard [67] to their providing.

From a security standpoint, SaaS providers are responsible for managing the data placement and computations in the cloud environment and to respect customer organizations' security policies and the SLAs [21][66]. Encryption solutions and well-thought-out approach to identity and access management will be essential to protect data in a cloud environment [67]. SaaS providers may be required to agree with audit requirements posed by customer organizations and third parties [66]. One scalable strategy for the SaaS provider is to make their audit reports available to all clients (under a non-disclosure agreement) [21]. These reports, proving compliance with clearly defined, international standards, would work as incentives for progressing past cloud migration [21].

5.6.5 Data partitioning

As mentioned, privacy is one of the most defining schemes of data partitioning, i.e. deciding data location in a hybrid cloud deployment. Enterprises desire to maintain sensitive data and processes within their internal network boundaries, whereas it could be beneficial to store less sensitive data in a public cloud [20][28] and run less sensitive processes in the cloud [7]. This is true for datasets, but also for *data projections*: it could be viable to create business requirement specific projections of datasets and entries and store the projections in the cloud [29], as sensitive entries or attributes could be excluded from these projections.

In addition to privacy concerns, cost and performance optimization are key factors when making decision about data location [14][18][28]. If throughput, latency and confidentiality are considered minor issues for a certain set of data, then storing it in a public cloud could be a cost saving solution [14]. In general, partitioning data over multiple clouds will

increase the application latency compared to a scenario in which the whole application resides within a single network [20].

If real-time data synchronization is not feasible, data could be partitioned by time: data with little or no demand for being real-time can be synchronized between environments in batches and/or asynchronously [14][29]. In hybrid cloud SaaS applications data generality could also be a viable partitioning scheme: data that is shared between tenants could be located in the public cloud and tenant-specific data on-premises [66].

5.6.6 Application partitioning

Migrating all legacy application components to a public cloud could be infeasible or it could end up being too expensive [28], making a hybrid approach more attractive. In a hybrid cloud deployment, networking will affect the overall performance, because the system is inherently distributed. It has to be decided which components are feasible to be located in the public cloud and which components should be located on-premises [20][28]. To make this decision, it is necessary to understand both the existing deployment models of the application and the behavior of the application's components [28]. In this context, component behavior consists of both the behavior within a component and the interaction between components [20][28].

According to Karthikeyan & Nandhini, when only some of the components of a legacy application are migrated into a hybrid cloud deployment, hidden optimization (i.e. ones that are manifested only when the component is a part of a monolithic single-environment setup) may have a major negative impact on the performance and scalability of the application [28]. Considering a green-field hybrid cloud SaaS application, connectivity with the existing legacy applications must be planned in advance: differences between technologies may cause significant refactoring, testing and need for reintegration with the legacy parts [28].

Not unlike in the case of data partitioning, optimization of cost and performance is a key consideration also from the point of view of application partitioning [28]. Locating *storage intensive* components near data storages that they interact extensively with reduces *wide area network* communication costs and response times [20]. On the other hand, compute intensive components should be located to an environment with sufficient computing resources [20].

5.6.7 Connectivity technology

In every hybrid cloud deployment, the issue of inter-cloud connectivity has to be overcome to allow secure communications for a system distributed across two or more networks [6][66][69]. As mentioned in subchapter 5.6.1, this challenge should be solved by the SaaS provider by whom the customers are given instructions and requirements about the means of connectivity [21]. From the customer's point of view, it would be beneficial to use as little separate connectivity technologies as possible, because an increasing amount of these technologies can lead to infrastructure fragmentation, device sprawl and duplication of integration processes [4]. A hybrid cloud deployment should be extensible and easy to integrate with on-premises systems [4].

In the literature, the following technologies were mentioned to solve the connectivity challenge at least partially and some solution models were discussed in depth:

- Service Bus (Enterprise Service Bus [62], Cloud Service Bus [77])
- VPN [4][6][7][20][77].

The following technologies were briefly mentioned to solve the connectivity challenge at least partially:

- API (Application Programming Interface) Management [62]
- iPaaS (integration PaaS) [62]
- EAI (Enterprise Application Integration) [62]
- REST API [29].

According to Chen *et al.*, VPN is a common solution for bridging private and public clouds together [6]. VPN is proposed as a solution model by Chent *et al.* [6] and Cheung [7] and is briefly mentioned as a solution model by Breiter & Naik [4], Hajjat *et al.* [20] and Zou & Deng [77].

A service bus is itself a complex application, consisting of several layers related to messaging, routing, monitoring and service registering [77]. It is proposed a solution model by Zou & Deng [77] and briefly mentioned as a solution model by Pathak & Khandelwal [62].

Azure's virtual network solution, *Azure VNET*, and the service bus based *Azure Service Bus Relay* are discussed in subchapters 6.2.1 and 6.2.2 respectively.

5.6.8 Performance

As mentioned, performance may be an issue in vertical hybrid cloud deployments. It is expected that, when requesting and transferring data across a wide area network, the throughput will be much less than in a local area network [20][28]. This is both due to

smaller bandwidth but also due to greater latency between separate distributed system components [28]. In their study Faul *et al.* present an empirical comparison of latencies between the two scenarios: in one scenario the application is located entirely in a single LAN, whereas in the other scenarios the application is distributed across a variety of cloud environments [14]. The results are not analyzed here in depth, but they clearly indicate that communication within a single environment is much more performant than communication between different cloud platforms or within a single cloud platform. The results highlight the importance of analysis of data and application partitioning. For better performance, components with high interdependency should be located near each other [20].

6. TECHNOLOGIES

In this chapter, MTHC enabling technologies are presented. These technologies are limited to what will be used or considered in the case examples, discussed in chapters 7 and 8, and by the organizational limitations described in the introduction chapter.

6.1 ASP.NET Core

ASP.NET Core is a cross-platform, open source redesign of ASP.NET, Microsoft's web development framework for building web apps on the .NET platform [49].

6.1.1 Dependency injection

ASP.NET Core has a built-in DI framework that makes configured services available to an application's classes [33]. In ASP.NET Core, *application services* are objects that are used by the application (e.g. logging service, data access service) [33]. A *dependency* is any object that another object requires [41]. Examine the following example. *SalesPersonProvider* is an application service that has a method called *GetSalesPerson* for getting an object of type *SalesPerson* that matches the given *salesPersonId*.

```
public class SalesPersonProvider
{
    private readonly IStorage storage;
    public SalesPersonProvider(IStorage storage)
        => this.storage = storage;

    public SalesPerson GetSalesPerson(string salesPersonId)
        => storage.QuerySalesPerson(salesPersonId);
}
```

SalesPersonProvider has a dependency on *IStorage*, which is an abstraction of a storage implementation that implements at least the method *QuerySalesPerson* that receives *salesPersonId* and returns the matching *SalesPerson* instance. Both *SalesPersonProvider* and *IStorage* can be configured as application services during application startup:

```
services.AddTransient<SalesPersonProvider>();
services.AddTransient<IStorage, SqlServerStorage>();
```

Once an application service has been configured, it can be *requested* from the *DI container* called *service provider*, as follows:

```
var spProvider = serviceProvider.GetService<SalesPersonProvider>();
var salesPerson = spProvider.GetSalesPerson("123");
```

Since *IStorage* is a dependency for *SalesPersonProvider*, it is also requested. In ASP.NET Core DI, each requested dependency in turn requests its own dependencies [41]. The DI container resolves the dependencies in the graph and returns the fully resolved application service [41]. The set of dependencies that must be resolved is typically referred to as a dependency tree*,* dependency graph or object graph [41].

In the example above, the application services were configured with a *transient* lifetime. ASP.NET Core DI has three options for lifetime: *transient, scoped* and *singleton*. Application services with a transient lifetime are instantiated each time they are requested, whereas singleton services are instantiated only once and the same object instance is provided for each request thorough the lifetime of the service provider [41]. Scoped services are instantiated once per client request [41] (e.g. once for each incoming HTTP request). As an example, *HttpContextAccessor* is one of the scoped lifetime application services that are provided out-of-box by the ASP.NET Core framework. During an HTTP request, if an application service requests *HttpContextAccessor* from the DI container, the resolved *HttpContextAccessor* instance will be able to provide an *HttpContext* instance for the requesting application service. *HttpContext* contains information about the current HTTP request context, such as the request *Uniform Resource Locator* (URL), HTTP method, HTTP headers and request body.

As discussed in chapter 3, Walraven *et al.* claim that support for tenant specific customization, i.e. the *ServiceInjector,* can be implemented with DI. In the simplistic example above, *SqlServerStorage* (=*ApplicationService*) will be resolved for *IStorage* (=*IApplicationService*), but any additional *TenantID* dependent logic can be added within this DI configuration. Consider Program 1:

```
    services.AddTransient<SqlServerStorage>();
    services.AddTransient<PostgreSqlStorage>();
    services.AddScoped<IStorage>(serviceProvider =>
    {
        // Resolve the current TenantID
        var currentTenantId = serviceProvider
            .GetService<HttpContextAccessor>().HttpContext
            .ResolveCurrentTenantIdFromHttpContext();

        // Resolve the ApplicationService
        if (currentTenantId == "tenantA")
            return serviceProvider.GetService<SqlServerStorage>();
        else if (currentTenantId == "tenantB")
            return serviceProvider.GetService<PostgreSqlStorage>();
        else
            throw new NotImplementedException();
});
```

Program 1. *ServiceInjector example with ASP.NET Core DI. The scoped IStorage can be resolved as SqlServerStorage or as PostgreSqlStorage, depending on the tenant.*

Now, if the resolved tenant is *tenantA*, then the resolved *ApplicationService* will be *SqlServerStorage*, and if *tenantB*, then it will be *PostgreSqlStorage*. This means that the *ServiceInjector*, as Walraven *et al.* described it, can be implemented with ASP.NET Core DI *if* the tenant can be resolved from the current *HttpContext* with some method like *ResolveCurrentTenantIdFromHttpContext().* Tenant resolution from the current *HttpContext* is discussed in subchapter 6.1.3.

6.1.2 Authentication

ASP.NET Core includes many application services within the framework. As an example, there are several application services for user authentication purposes. In ASP.NET Core, authentication functionality is configured as an application service with ASP.NET Core DI and *enforced* with a *middleware*. The ASP.NET Core request pipeline consists of a sequence of request delegates called one after the other [34]. These request delegates are called *web application request middleware*, or just *middleware*. Figure 8 demonstrates the concept. The thread of execution follows the arrows.

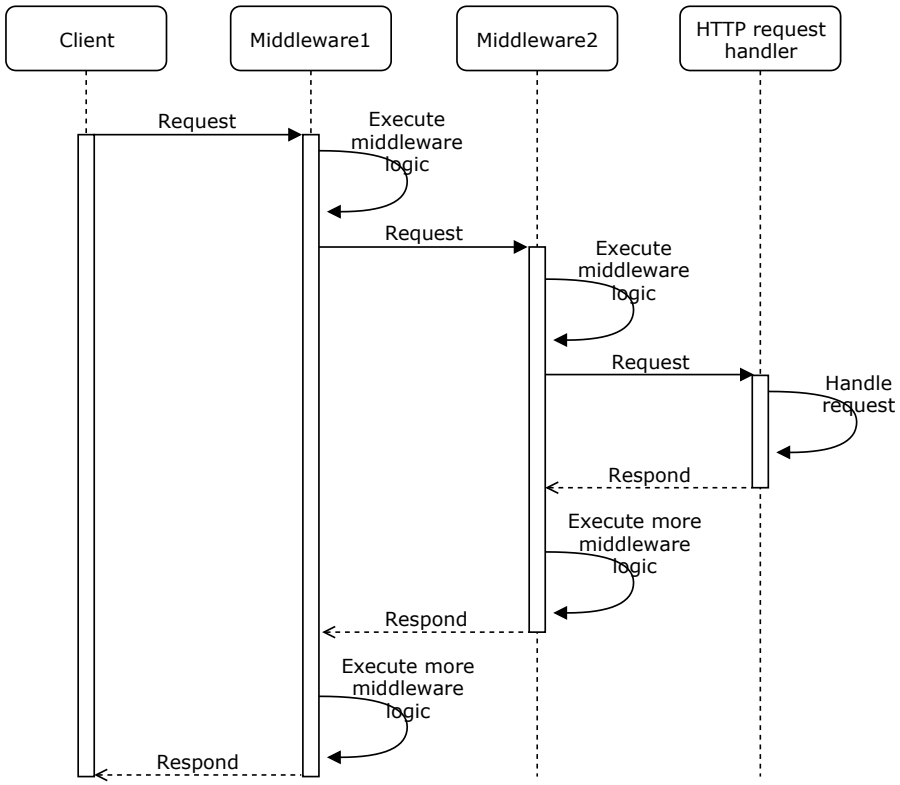

Figure 8. *ASP.NET Core middleware pipeline. Each middleware has access to the incoming HTTP request and the outgoing HTTP response and can perform middleware specific application logic. Together the pieces of middleware form a middleware pipeline.*

Each middleware can perform middleware-specific actions before and after the next one [34]. *UseAutentication* middleware adds authentication to the request pipeline, and if authentication fails, the request does not get further in the pipeline.

The framework provides several *authentication schemes* that can be used to configure the authentication service. *JWT Bearer authentication scheme* configures the authentication service to require a valid JWT from each incoming HTTP request. A JWT is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a *JavaScript Object Notation* (JSON) object that is used as the payload of a JSON Web Signature structure or as the plaintext of a JSON Web Encryption structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code and/or encrypted [26].

OpenID Connect authentication scheme allows authentication service to enforce OpenID Connect based authentication. OpenID Connect is a simple identity layer on top of the old OAuth 2.0 protocol. It allows client applications to verify the identity of the user based on the authentication performed by an external identity authority, as well as to obtain basic profile information about the user in an interoperable and REST-like manner. OpenID Connect allows client applications of all types, including Web-based, mobile, and JavaScript clients, to request and receive information about authenticated sessions and end-users. [60]

OpenID Connect authentication scheme assumes that there is an external OpenID Connect authentication flow supporting authentication service (such as IdentityServer4 [24] or a social authentication service such as Azure Active Directory or Google Identity) extending SSO capabilities to the ASP.NET Core application. On the other hand, *JWT Bearer authentication scheme* requires only that some validation process for the incoming JWT token has been configured. This means that the validation can be performed, for example, using an external authentication service, functioning as the *identity authority*, or using an in-process identity management. To add an in-process identity management to an ASP.NET Core application, *ASP.NET Core Identity* service must be configured with DI.

ASP.NET Core Identity is a membership system that adds login functionality to ASP.NET Core applications [45]. User accounts and their login information can be stored either in an ASP.NET Core Identity configured storage or an external login provider can be configured [45]. ASP.NET Core Identity can be configured using, for example, a *SQL Server* database to store usernames, password hashes, and profile data. Alternatively, any other custom implementation can be used [45]. The possibility to use a custom identity

storage implementation is beneficial regarding the MTHC architecture, as master systems often have their own existing identity management systems in place and the add-on applications are required to comply with the existing authentication methods.

6.1.3 Finbuckle.MultiTenancy

In addition to implementing the *ServiceInjector* with ASP.NET Core DI, there are few open-source multitenancy enablement layer libraries available for ASP.NET Core. On one hand, there are comprehensive framework-like solutions such as *ASP.NET Boilerplate Framework* [1] and *cloudscribe* [9], but these tend to dominate the total application architecture and be very biased in that. *Finbuckle.MultiTenancy* is one of the few *low intrusion* multitenancy enablement libraries for ASP.NET Core.

Not unlike ASP.NET Core Authentication, *Finbuckle.MultiTenancy* is configured as an application service with ASP.NET Core DI and enforced with a middleware. When configuring the application service, a *tenant resolution strategy* and a *tenant configuration storage accessor* must be provided [15]. A tenant resolution strategy describes how the requesting tenant should be resolved. As an example, one could receive the claimed *TenantID* from the current HTTP request's URL's subdomain or from a claim in the JWT in the request Authorization header. Tenant configuration storage accessor is used to access the tenant-specific configuration using the resolved *TenantID* as the key.

The model provided by *Finbuckle.MultiTenant* is very similar to the model proposed by Walraven *et al.* (discussed in chapter 3). As a concept, *tenant resolution strategy* is equivalent to MTEL. Tenant configuration storage accessor, i.e. the *ConfigurationManager* in the model proposed by Walraven *et al.*, provides instances of *Configuration*. In addition, *Finbuckle.MultiTenancy* provides methods for extracting a *MultiTenantContext*, i.e. *TenantContext*, from the current *HttpContext* in any application service. *Finbuckle.MultiTenancy* does not inherently support multi-tenant customization, i.e. resolving different *ApplicationServices* for each *IApplicationService* per-tenant basis, but, as already discussed, this can be performed with plain ASP.NET Core DI.

6.2 Azure

Microsoft Azure is a cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers. It provides SaaS, PaaS, IaaS and supports many different programming languages, tools and frameworks, including both Microsoft-specific and third-party software and systems.

6.2.1 Azure Virtual Network

Azure provides *Azure Virtual Network* (VNET) IaaS for building virtual private networks in Azure. VNET allows many types of Azure resources, such as Azure Virtual Machines, to securely communicate with each other, the internet, and on-premises networks. VNET is like traditional VPNs that one would operate in one's own data center, but brings along additional benefits of Azure's infrastructure, such as scalability, availability and isolation. [52]

There are three options for connecting on-premises resources to a VNET: *Point-to-site VPN connection, Site-to-site VPN connection* and *Azure ExpressRoute* [52]. Point-to-site VPN connection establishes a secure connection between the VNET and an individual client computer, and it is useful for telecommuters, i.e. people who want to connect to Azure VNETs from a remote location, such as from home or a conference [56]. Site-to-site VPN connection is established between customer's on-premises VPN device and Azure VNET [52]. This connection type makes it possible for any authorized on-premises process to access the hybrid cloud's virtual network [52], which makes it suitable for hybrid cloud scenarios. The communication between the on-premises VPN device and the VNET is performed through an encrypted tunnel over the internet [52]. *ExpressRoute* makes it possible to extend on-premises networks into Microsoft's cloud services, such as Azure, Office 365, and CRM Online [52].

When creating a VNET, a custom private IP *address space* must be specified for the VNET to use [52]. For example, if a VNET was created with a block 10.0.0.0/16, then the VNET address space would include IP addresses 10.0.0.0-10.0.255.255. By convention, VNET address space must be further divided into *subnets*, to which Azure resources can be deployed. If, for example, a subnet with block 10.0.128.0/24 was created to the mentioned VNET address space, then a resource such as a virtual machine could be deployed in it, and the virtual machine would be assigned a private IP like 10.0.128.4.

According to Microsoft, one of the best practices of VNET planning is to make sure ahead of time that the chosen subnet IP address range does not overlap with other network ranges of the client organization [52], i.e. the customer environment. In an MTHC scenario, it can be difficult to plan this in advance because each tenant may have very heterogenous internal network setups and reserved IP address ranges, and new tenants are added to the system over time.

6.2.2 Azure Relay Hybrid Connection

Azure Relay (or *Azure Service Bus Relay*) makes it possible to securely expose services that run within a corporate network to the public cloud. Unlike Azure VNET, this can be

done without opening a port in the corporate firewall or making other intrusive changes to the corporate network infrastructure. VNET and other VPN solutions are used for *infrastructure level* resource exposing (e.g. ports on a machine), whereas Azure Relay exposes only *application level* listeners, such as C# methods [51]. Having access to only application level listeners can make Azure Relay more secure, as the access can be limited to only a very specific set of functionalities, but it is also a limitation since any arbitrary HTTP endpoints cannot be directly requested without making the listeners to explicitly support this.

According to Microsoft, Azure Relay can be used for traditional one-way request/response communication; event distribution to enable publish/subscribe scenarios; and bi-directional and unbuffered socket communication across network boundaries [51]. Azure Relay is based on the *relayed data transfer pattern*, which involves the following steps [51]:

1. An on-premises service connects to the relay service through an outbound port
2. It creates a bi-directional socket for communication tied to a URL (provided by Azure Relay)
3. The client can then communicate with the on-premises service by sending traffic to the relay service targeting that URL
4. The relay service then relays data to the on-premises service through the bi-directional socket dedicated to the client. The client does not need a direct connection to the on-premises service. It does not need to know the location of the service and the on-premises service does not need any inbound ports open on the firewall.

Azure Relay Hybrid Connection is a secure, open-protocol evolution of Azure Relay and abstraction on top of Azure Relay. It allows sending requests and receiving responses to/from Azure Relay using HTTP(S) or web sockets [51]. In the MTHC scenario this would mean that the on-premises master system could be sent requests via HTTP from add-on applications running in the public cloud and the master system could respond back.

6.2.3 Azure App Service

Azure App Service is an HTTP-based PaaS for hosting web applications, REST APIs, and mobile backends. Azure App Service supports several different development technologies, be it .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python. Being PaaS, it provides many utilities for making application development and hosting easier, such as load balancing, autoscaling, and management automation. [32]

In App Service, each web application runs in an *App Service Plan*. The compute resources used by the web applications are defined by their App Service Plans. These compute resources are analogous to the server farms in conventional web hosting. One

or more applications can be configured to run on the same computing resources, i.e. within the same App Service Plan. [36]

App Service includes *App Service Hybrid Connections* within the platform. As described, Hybrid Connection is also a separate feature within Azure Relay service. Within App Service, Hybrid Connections provide access from the applications running on the App Service Plan to resources in other networks. Figure 9 displays the basic setup for App Service Hybrid Connections. [35]

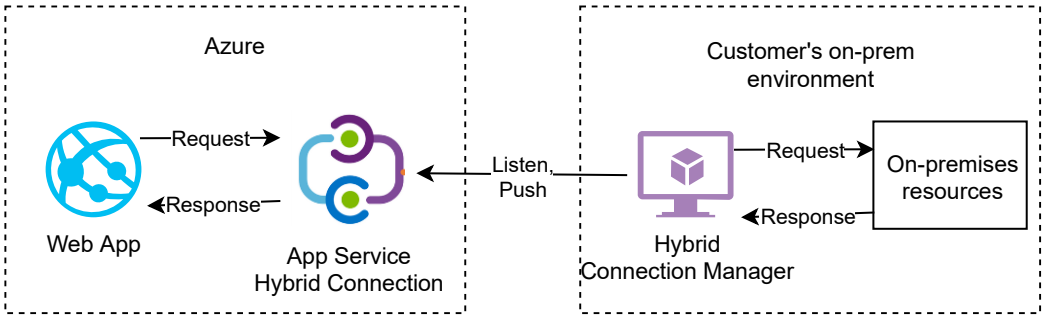

Figure 9. *To allow the Web App to access on-premises resources using App Service Hybrid Connections, Hybrid Connection Manager is required to be installed on-premises.*

When Hybrid Connections are enabled for App Service, an instance of Azure Relay is allocated, because App Service Hybrid Connections uses Azure Relay for indirect communication. This process is invisible to the App Service user. Unlike Azure Relay, App Service Hybrid Connections require *Hybrid Connection Manager* to be installed to a Windows machine in the on-premises network. The instance of Azure Relay that get allocated relies on web sockets for connectivity. Web sockets are only available on Windows Server 2012 or later. Thus, Hybrid Connection Manager is not supported on anything earlier than Windows Server 2012. [35]

6.3 Azure Pipelines

Azure DevOps is a successor to Microsoft's *Visual Studio Team Services* which is the online version of Team Foundation Server. Originally a source code management tool where development teams could share and work on code collectively, Azure DevOps has grown to become a platform where development projects can be managed, tested, built and released.

Part of Azure DevOps providing, *Azure DevOps Pipelines* or just *Azure Pipelines* is a service that combines *Continuous Integration* (CI) and *Continuous Delivery* (CD) to constantly and consistently test and build code and ship it to any target. In Azure Pipelines, *build pipelines*, e.g. CI pipelines, test and build code and produce deployable *artifacts*,

which include both infrastructure and application binaries. *Release pipelines* consume these artifacts to deploy new versions to the configured *targets*. Targets can include container registries, virtual machines, Azure services and any on-premises or cloud environments. [50]

In Azure Pipelines, an *Azure Pipelines Agent*, or just *agent,* is an installable piece of software that can run *jobs*, each job consisting of *tasks* [39]. For example, a build agent can be installed on a virtual machine and the build agent can then be used to run jobs, such as building or testing code. The build agent downloads code to the virtual machine, performs the job and uploads the job results (e.g. binaries or test results) back to Azure Pipelines. Agents can also be used for deploying release artifacts, such as binaries. When an agent is used to deploy artifacts to a server or a set of servers, it must have "line of sight" connectivity to those servers [42]. For example, if an artifact is desired to be deployed to an on-premises environment that resides behind a private network firewall, the deploy agent must first be installed to the private network on a server that allows outbound connections to the Azure Pipelines service over the internet [42]. Figure 10 displays this setup. As shown in the figure, the on-premises deploy agent must have a "line of sight" to both Azure Pipelines and to the target on-premises servers.

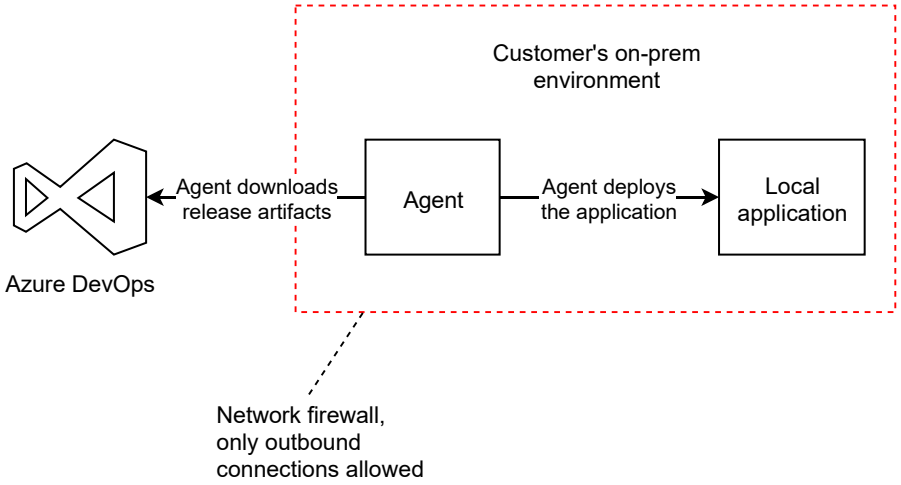

Figure 10. *Azure Pipelines on-premises deploy agent setup. The agent installed on-premises can download release artifacts from Azure DevOps Pipelines using an outbound connection.*

Azure Pipelines Agents have several installation prerequisites. If installed on a machine with Windows 7 or Windows 8.1 or Windows Server from 2008 R2 SP1 to Windows Server 2012 R2, the machine must also have at least PowerShell 3.0 or higher and .NET Framework x64 4.5 or higher. For Windows 10 and Windows Server 2016 and higher there are no prerequisites. [38]

7. CASE: DQM TOOL

DMS is a system included in the company's dealership management service offering. It combines personal data from several data sources, both internal and external. These different versions of personal data are first stored in the *personal data storage*. A scheduled batch job attempts to aggregate these pieces of data into combined customer entries. Figure 11 displays the overall data flow from data sources to DMS.

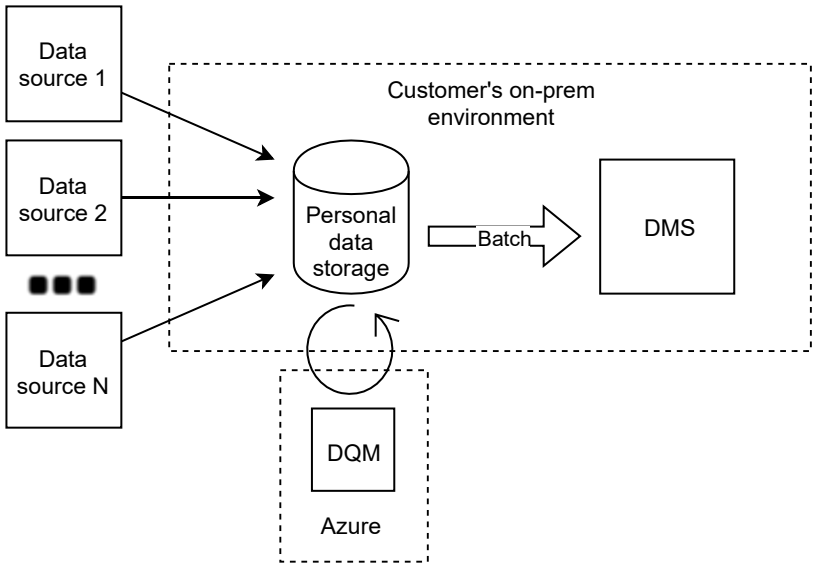

Figure 11. *Data flow to DMS. Personal data storage contains personal data from several data sources. A batch job attempts to aggregate these pieces of data into combined customer entries for DMS to consume. Since the batch job does not produce accurate results all the time, DQM is designed to complement it.*

The aggregation that the batch process performs is based on heuristics that do not always end up with the desired result due to unexpected corner cases and flaws in the source data. Duplicates and false matches end up to DMS. Therefore, manual data processing efforts are required for taking care of these exceptions. A new DQM tool is designed to assist a knowledgeable user in these manual tasks.

DMS is an OTS/ASP, meaning that instances of personal data storage and DMS have been installed on each customer's premises, as is displayed in Figure 12. Therefore, the MTHC architecture can be a viable option. In this case, DQM would be the *add-on application* and DMS would be the OTS/ASP *master system*.

Personal data storage is an *IBM DB2 for i* database without any HTTP APIs available for data access. In addition, each customer has a Windows Server (at least 2008 R2) available if any on-premises deployment is required. These Windows Servers have the IBM DB2 for i *Open Database Connectivity* (ODBC) drivers readily installed.

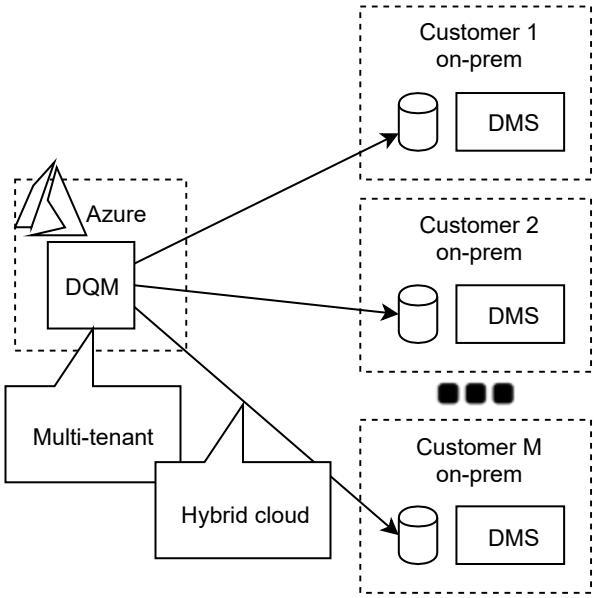

Figure 12. *A sketch of the MTHC architecture for DQM. DQM would be a multi-tenant add-on application accessible over the internet. Hybrid cloud connectivity would be required in order to access the personal information storages on each customer's premises.*

IBM DB2 for i is an IBM's implementation of the DB2 RDBMS, which comes with IBM i operating system. IBM i runs on IBM's Power Systems and Pure Systems for minicomputers and enterprise servers. It is targeted for mid-sized to large enterprise businesses and is designed for ease of use, easy deployment and maintenance of reliable server operations. [64]

ODBC is a widely accepted API for database access. It uses SQL as its database access language. ODBC is designed for maximum interoperability, i.e. the ability of a single application to access RDBMs with the same source code. Database applications call functions in the ODBC interface, which are implemented in database-specific modules called drivers. [47]

7.1 MTHC design challenges

There were three design challenges that turned out to be major issues considering a potential MTHC architecture. These issues were *compliance to regulations*, *connectivity* and *authorization*.

Having customer data processed and possibly metadata and application logs stored in a public cloud caused issues regarding *compliance to regulations*: The personal data storage contains personal data combined from several different data sources with their own regulations and service agreements.

Connectivity turned out to be a slight issue, as there is no HTTP API directly available for executing remote procedure calls to the personal data storage. This means that either the IBM DB2 for i ODBC driver would have been necessary to be installed to the Azure App Service PaaS platform interface or a separate HTTP API application would have to be deployed on each customer's premises, that would translate HTTP calls from the DQM application running on Azure App Service to SQL queries. Installing the ODBC drivers on Azure App Service PaaS is impossible due to the high level of service abstraction of the platform. Another option would have been to install the drivers on a dedicated virtual machine and use that machine as a multi-tenant gateway, but that would have introduced additional infrastructure that should have been maintained, which was not desirable.

In addition, one requirement for DQM was that the users should be authenticated with their internal Windows Active Directory authentication and authorized based on whether a user belongs to a local Active Directory group or not.

7.2 Alternative solution and evaluation

Due to the deal breakers mentioned, an MTHC approach was not fully applied in the case of DQM. All three issues were resolved by having DQM installed locally on customer's premises: data stays within organizational network and ODBC drivers Active Directory are readily available. Figure 13 displays the final delivery model that was implemented. Unlike required by Goal 4, there is no multi-tenant application instance, but instead the build and release pipelines for each customer are based on the same Azure Pipelines *tasks* and merely configured per-customer. This means that, even though a shared multi-tenant instance was not a viable solution, reusability was achieved in the level of build and release pipelines.

When a new version of DQM is released with Azure DevOps Pipelines, the agent in each customer's environment downloads the release artifacts and performs the deployment job: installs the new version of DQM on the local *Internet Information Services* (IIS) and performs smoke tests. IIS is a Web Server for Windows Server. Therefore, DQM can still benefit from CI/CD, even though the application is not running on Azure App Service. Hence, Goals 1 and 2 were reached.

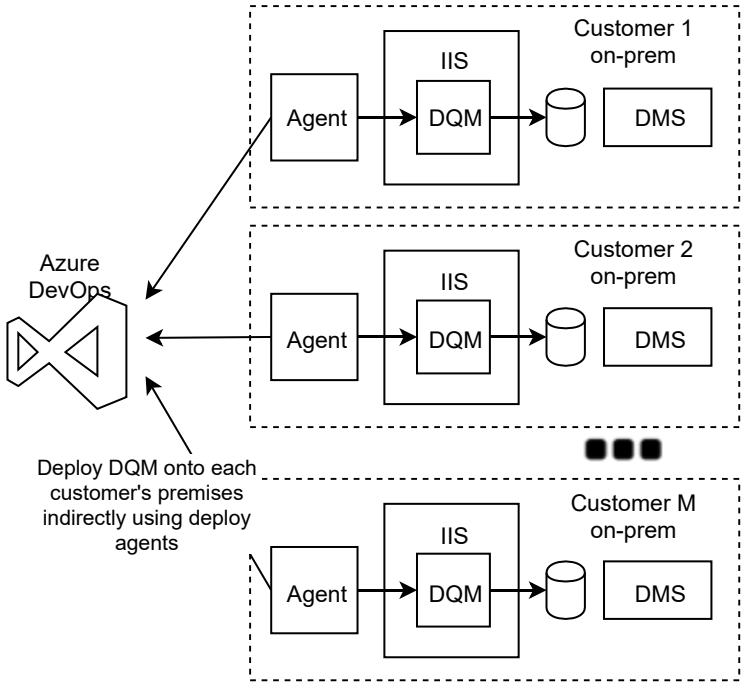

Figure 13. *The final deployment model of DQM. DQM add-on application is run on IIS locally on each customer's premises. Deploy agents are installed in these same environments, which enables continuous delivery.*

There are a few downsides to this. Azure Pipelines Agent, IIS and .NET Core Runtime must be installed in each customer's environment, which to some extent is a manual process. .NET Core Runtime is a system-wide installation of .NET Core framework required by .NET Core applications. Depending on the environment, these prerequisites may have some prerequisites of their own, such as those for Azure Pipelines Agent mentioned in subchapter 6.3. These problems make the initial deployment to each customer's environment a manual and therefore non-scalable process, as unpredictable environment specific issues must be resolved. In addition, changes in customer infrastructure may require performing these manual installation steps again. Instructing customers themselves to do these installation steps as a self-service could be an option to consider, but customer environment specific peculiarities render it extremely difficult to create an exhaustive set of instructions. Therefore Goal 3 was not fully reached, as including new customers requires some manual efforts.

DQM application itself has very little requirements for variability per customer, the only few variability points being infrastructure related configuration, such as connection string to the database. These configuration settings are stored in Azure Pipelines as plaintext or as secrets, depending on the confidentiality of each setting, and included to the build during release. Another option would be to store configuration as environment variables to each customer's environment, but this would make it harder to update and version control them, as they would be only key-value-pairs within the environments themselves.

8. CASE: TIRE HOTEL PDA

The company's dealership software service portfolio also includes an OTS/ASP tire hotel system, THS. THS is used by certain car dealer and maintenance businesses that offer tire hotel services. This means that they offer seasonal tire changing services and storage for the changed tires. Previously, the total process of changing tires; transportation between workshops and storages; storage; and storage organization involved cumbersome manual steps in order to track the status of each set of tires. Therefore, THPDA is designed to quicken this process. In the minimum-viable-product stage, THPDA should streamline the process by allowing tire sets to be tracked by merely reading barcodes that have been put on each of them. Figure 14 illustrates this total process.

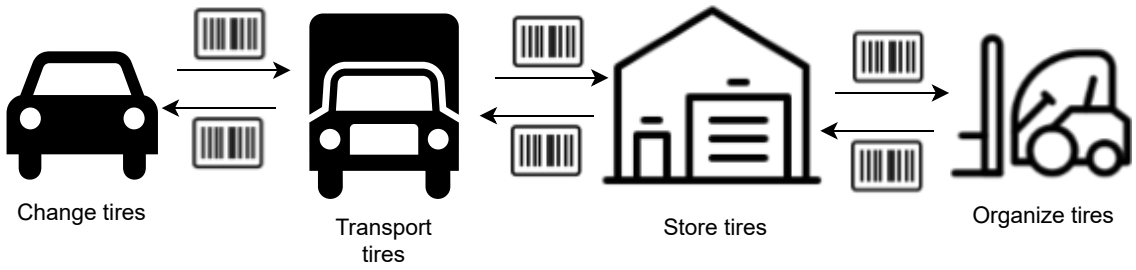

Figure 14. *The tire hotel process. In a nutshell, customers bring their cars in for tire changing. After the tires have been changed, they are transported to storages. While stored, tires may be reorganized. The barcode icons indicate the points when barcodes are read from tires.*

THS is run in the exact same environment as DMS using the OTS/ASP deployment model. THS owns the state of each set of tires in its internal storage and provides an HTTP API for querying and altering these states. Figure 15 shows an overview of the THS deployment.

As shown in the figure, THS, like many legacy systems, use its own identity management system called ASUSER. Specifically, ASUSER is an identity management system for the vehicle maintenance and after-sales family of dealership portfolio systems, containing user identities for vehicle maintenance, transport and storage personnel.

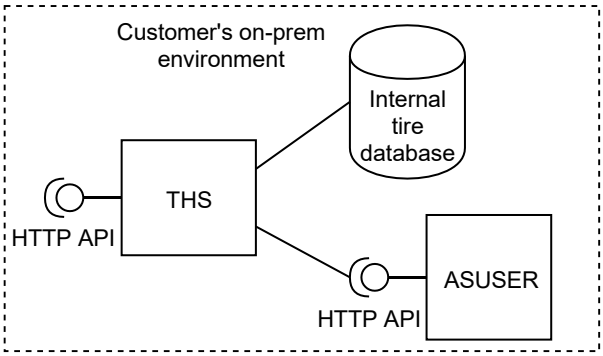

Figure 15. *Overview of the THS deployment. THS itself provides an HTTP API and uses the HTTP API of the ASUSER identity management system for user authentication. THS stores tire sets' states in an internal database.*

The PDA device used for THPDA is Zebra TC-25. By default, it has Android 7.1.2 and supports Wi-Fi and 5G internet connection.

If THPDA is considered an *add-on application* and THS is considered the *master system*, the MTHC architecture could be a viable option, especially since the PDA devices could access a "THPDA web application" over the internet via browser or an Android hybrid application. Figure 16 shows a sketch of this MTHC architecture.

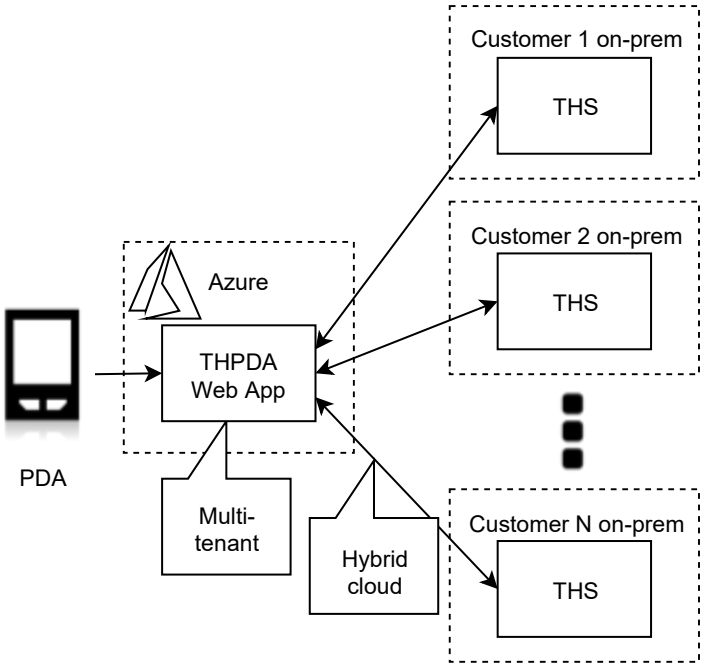

Figure 16. *A sketch of the MTHC architecture for THPDA. A multi-tenant THPDA Web App would provide the backend for the user interface used via the PDA devices. Accessing THS on each customer's premises would require hybrid cloud connectivity.*

THPDA Web App would be the multi-tenant add-on application. Being a hybrid cloud deployment, THPDA Web App could access each customer's THS and thus allow changing tire sets' statuses as barcodes are read with the PDA devices.

8.1 MTHC design challenges

In order to change tire sets' statuses, THPDA Web App would need to have connectivity from Azure to each customer's environment. The customer organizations of dealership solutions vary a lot in size and in IT capability, so the connectivity solution should be non-intrusive and easy to set up. The presented connectivity options provided by Azure, Azure VNET and the Hybrid Connection solutions, differ in the ease of implementation and intrusiveness. The Hybrid Connection solutions do not require enterprise firewall configuration efforts, unless its required outbound ports are blocked, which is unlikely. Unlike Azure VNET, Azure Relay does not require reserving a subnet of IP addresses for the application, which could very likely lead to collisions with other subnets as new customer organizations with their own infrastructure setups subscribe to THPDA (as was discussed in subchapter 6.2.1), rendering infrastructure management unnecessarily complex.

THPDA Web App requires access only to THS HTTP API, meaning that having *application level access* to a listener that forwards incoming HTTP requests to THS HTTP API would be sufficient. This makes the Hybrid Connection based solutions more desirable, since they provide application level access, whereas Azure VNET provides infrastructure level access, as was discussed in subchapter 6.2.1.

Comparing the two Hybrid Connection solutions, Azure Relay Hybrid Connection and App Service Hybrid Connections, the former would require implementing a separate listener application and deploying it on each customer's premises, which implies additional development efforts. The latter would require installing Hybrid Connection Manager on each customer's premises, which is not viable, because Hybrid Connection Manager has several prerequisites that some of the customers are unable to fulfill with a reasonable amount of effort and investment. Therefore, Azure Relay Hybrid Connection was chosen as the connectivity technology for THPDA. In the scope of the project, the on-premises process that runs the application level access enabling listener is called *OnPremGateway*.

As mentioned, Azure Relay Hybrid Connection is based on Azure Service Bus, meaning that the messaging between OnPremGateway and THPDA Web App in Azure will be asynchronous. Compared to Azure VPN, requesting is more expensive as the volume of

requests and the size of each request increase, and less performant. This is not an issue in THPDA as the expected volume of requests (the number of barcodes read) should be relatively low and the payloads of each request are very lightweight. There are also no strict requirements for request latencies, but, if the latencies end up being too large, the client-side processing can be refactored to be asynchronous instead of being strictly synchronous (i.e. the PDA user would not have to wait for each request to complete). An overview of the selected Azure Relay Hybrid Connection based approach is displayed in Figure 17.

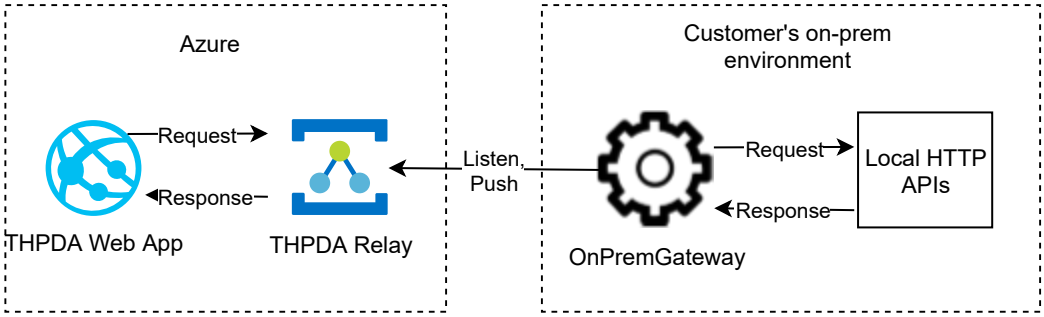

Figure 17. *THPDA Relay Hybrid Connection setup. THPDA Web App, which provides the backend for the PDA frontend, can access on-premises APIs indirectly using Azure Relay Hybrid Connections and the instance of OnPremGateway that should also reside on-premises. The instance of Azure Relay used by THPDA Web App is called THPDA Relay.*

As shown in the figure, all requests from THPDA Web App are targeted to THPDA Relay, which is an instance of Azure Relay with a Hybrid Connection assigned for each customer. OnPremGateway is a Windows Service installed on-premises, which listens THPDA Relay for new messages. As new messages appear, OnPremGateway requests the local HTTP APIs (i.e. THS HTTP API). After getting a response for a request (e.g. whether changing the state succeeded or not), OnPremGateway pushes the response to THPDA Relay which gets forwarded to THPDA Web App as a response for the original request.

THPDA requires no own data storages because all state is stored in THS. This means that *data partitioning* or *storage variability* are no issues in this application, as no data is stored and therefore it is not necessary to be partitioned across the hybrid cloud, and due to storage being completely owned by the OTS/ASP master system, with the exact same storage model being provisioned for each customer. Considering that Azure Relay Hybrid Connection is used for connectivity, *application partitioning* is still something to consider, as one part of this distributed system resides in THPDA Web App in Azure and the other part in OnPremGateway on-premises. This partitioning problem can be solved by making THPDA Web App capture all business logic and the details of local HTTP

APIs and by making OnPremGateway merely a proxy. OnPremGateway forwards HTTP requests to the configured local HTTP endpoints and pushes the responses back to the configured THPDA Relay. This means that even if the local HTTP APIs change and evolve, only THPDA Web App must be changed and redeployed. Naturally, if the on-premises infrastructure changes drastically, OnPremGateway must be also redeployed.

Like with the DQM project, variance among customer environments is an issue with THPDA project as well, because OnPremGateway must be installed on-premises. Azure Pipelines Agents were installed in customer environments already for DQM and they can be used for deploying OnPremGateway as well, because Azure DevOps allows sharing agents between projects. OnPremGateway itself has certain prerequisites, such as a newer version of .NET Core Runtime that was installed for DQM. Fortunately, the runtime prerequisite problem can be eliminated by compiling OnPremGateway as a *self-contained deployment*. Unlike a traditional *framework-dependent deployment*, a self-contained deployment does not depend on the presence of shared components, such as .NET Core Runtime, on the target system [46]. All shared components are included with the application and are isolated from other .NET Core applications [46]. Self-contained deployment does lead to larger deployment sizes and does still depend compiling to target the correct operating system and processor architecture [46], so these have to be addressed build-time. Since OnPremGateway does not need have HTTP interfaces, but, being a listener, needs to be long-running, it can be executed without IIS as a Windows Service. This renders IIS, too, no longer a prerequisite.

Considering *authorization,* one functional requirement for THPDA was that its users should be able to log in to THPDA Web App using the same credentials as they use for THS. This means that ASUSER should be the identity authority for THPDA. As discussed in subchapter 5.6.2, authorization is one of the biggest challenges in hybrid cloud systems, and the security mechanisms, such as authentication, often are not cloud-ready [21]. This is true also for THS: as an example, there are no strict complexity requirements for passwords, and therefore there is a risk for user accounts to get compromised. In addition, even though THPDA Web App is an Azure App Services based web application in the public network, it is not desirable to authorize anyone who is not accessing the application using the browsers on the exact PDA devices. These two problems require an additional level of authentication. One solution would be to have the barcode readers and THPDA Web app in same VPN, rendering any external access impossible. Another less intrusive solution would be to use *client certificate authentication* on top of user authentication. It is an authentication method used to authenticate clients during the *secure*

sockets layer handshake with X.509 certificates. A negative aspect of using client certificate authentication is that it requires client certificate installation on each PDA device, but in the scope of this project this is a reasonable prerequisite, as there are also other unrelated manual configuration tasks that must be performed on the devices before production use. Since ASUSER provides an HTTP API for checking user credentials (as shown in Figure 15), THPDA Web App can access it via OnPremGateway similarly as it accesses THS HTTP API (as shown in Figure 17).

ASP.NET Core DI and *Finbuckle.MultiTenant* library were used for *tenant isolation*. This was mainly due to lack of other strong options, as was discussed in chapter 6.1.3. As mentioned, the current *TenantContext* can be resolved with any arbitrary tenant resolution strategy. In THPDA Web App, the tenant should be resolved from the incoming HTTP request's client certificate thumbprint during login and from the incoming HTTP request's JWT bearer token (authorization header) after login. Client certificate authentication is used only during login, because the way the user should choose the correct client certificate depends completely on the user's browser. For example, Google Chrome asks for the client certificate only once, whereas Firefox seems to ask it for all requests. Therefore, by requiring the client certificate only during login, the client certificate will be asked only during login and not for all subsequent requests after login, which would impair the usability of the user interface. Figure 18 displays these processes.

Please note that all layers of hybrid connectivity have been removed from the figure for brevity. After the client certificate has been validated, the tenant is resolved using the certificate thumbprint. The certificates installed on the PDA devices are tenant specific, so each tenant has its own certificate and therefore a certificate thumbprint. Only after the certificate is validated and the tenant is resolved, the incoming username and password are checked using the ASUSER instance on the very tenant's premises that was resolved. This way client certificate works also as a technique for achieving tenant isolation: the same login screen can be used for all tenants, because the sent client certificate is tenant specific and thus specifies which tenant's user is attempting to log in.

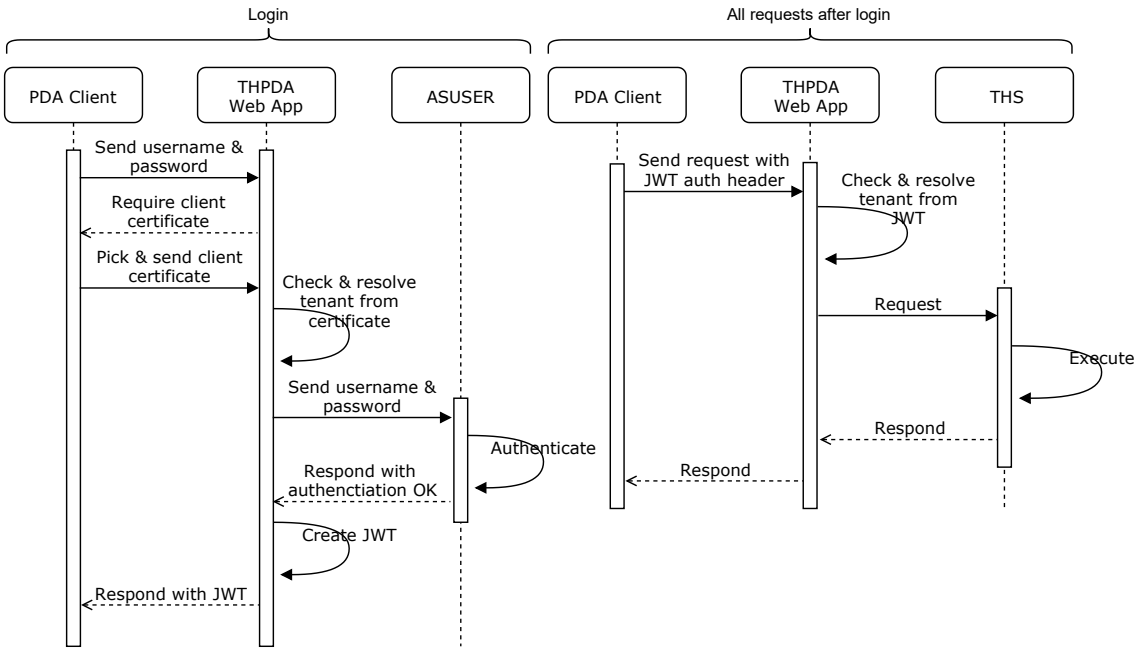

Figure 18. *THPDA authentication and tenant resolution during and after login. On the left-hand side, the login-time tenant resolution and user authentication are displayed. On the right-hand side, the authentication process during all subsequent requests is displayed. During login, the user credentials are checked using ASUSER on the resolved tenant's premises. During all subsequent requests, authentication is performed by inspecting the incoming JWT bearer token.*

If login succeeds, the client is responded with a JWT token. The client must attach this JWT token as a bearer token for each oncoming request after login. The token contains the *TenantID* that can be used for *TenantContext* resolution for each subsequent request. This can be achieved with a *Finbuckle.MultiTenancy* library's *Delegate tenant resolution strategy*:

```
services.AddMultiTenant().
    WithDelegateStrategy(context =>
    {
        var tenantId =
            GetTenantIdFromRequestJwt((HttpContext)context).Request);
        return Task.FromResult(tenantId.ToString());
    })
```

If a *TenantID* can be extracted using one of the configured strategies, then the library maps the *TenantID* into the corresponding *TenantInfo* object [16] (which represents the current tenant's *Configuration* in *Finbuckle.MultiTenancy*). The *TenantInfo* objects for each tenant are stored in a *multi-tenant store*, such as *in-memory store*, which holds all *TenantInfo* objects in the application memory. An *in-memory store* must be configured as follows:

```
services.AddMultiTenant()
    .WithInMemoryStore(Configuration
        .GetSection("InMemoryStoreConfig"))
```

This way *TenantInfo* objects can be stored using *ASP.NET Core Application Configuration* (AppConfig). *Finbuckle.MultiTenant* library requires a specific format from the AppConfig section that is used as a source for *TenantInfo* objects. For example, if a piece of JSON is used as an AppConfig source, then the *InMemoryStoreConfig* section should be formatted as shown in Program 2.

```
{
  "InMemoryStoreConfig": {
    "TenantConfigurations": [
      {
        "Id": "unique-id-for-tenant1",
        "Identifier": "tenant1",
        "Name": "Tenant 1 Organization Name"
        "Items": {
            "ClientCertThumbprint": "abc123",
            "RelayHybridConnectionString": "connstring&key={Key}",
        }
      },
      {
        "Id": "unique-id-for-tenant2",
        "Identifier": "tenant2",
        "Name": "Tenant 2 Organization Name"
        "Items": {
            "ClientCertThumbprint": "def456",
            "RelayHybridConnectionString": "connstring&key={Key}",
        }
      },
    ]
  }
}
```

Program 2. *TenantInfo configuration example in JSON format. The configuration includes two tenants, each having their dedicated client certificate thumbprints and Azure Relay Hybrid Connection connection strings. The configuration section is called InMemoryConfig, because the tenant configurations inside it will be loaded and cached in-memory during application start-up. The configuration data is stored in-memory as TenantInfo objects.*

If the current *TenantID* is one of the configured identifiers, then a *TenantInfo* object can be extracted from the current *HttpContext* in any application service. In the minimum-viable-product stage of THPDA, THPDA Web App contains only one application service that requires the *TenantInfo* object: *TenantHybridConnectionClientFactory*. *TenantHybridConnectionClientFactory* is used to build *HybridConnectionClient* objects, which are used to communicate with the OnPremGateway instances on each tenant's premises via THPDA Relay (see Figure 17). As opposed to THPDA Web App, OnPremGateway contains a *HybridProxyListener* object, which listens to the tenant's Hybrid Connection in

THPDA Relay and forwards requests to the local HTTP APIs (THS, ASUSER) as they are produced by *TenantHybridConnectionClient* in THPDA Web App.

Using *Finbuckle.MultiTenant* library's tenant resolution strategies and having a dedicated Hybrid Connection for each tenant together resolve the challenge of tenant isolation from the point of view of security. It does not address *tenant interference*. In THPDA, the volume of traffic is expected to be relatively low, consisting only of lightweight tire set status change commands, so performance and therefore tenant interference are not expected to be issues. If performance turns out to be an issue, the App Service Plan can be scaled up. In addition, the amount of variability is low, the only variation points being the tenant specific sections in AppConfig for client certificate and the Relay Hybrid Connection connection string. If configured or implemented incorrectly, these could potentially cause tenant interference issues, but the risk for this can be minimized with testing and test automation.

The data processed by THPDA is non-confidential and, excluding AppConfig and application logs, no tenant specific data is stored in cloud. Configuration and logs are only accessible by developers participating the project, and application logs do not store any personal information, except for username. Other than legal regulations for personal information management, there are no requirements for *compliance to regulations*. Figure 19 shows the MTHC architecture of the overall THPDA solution.

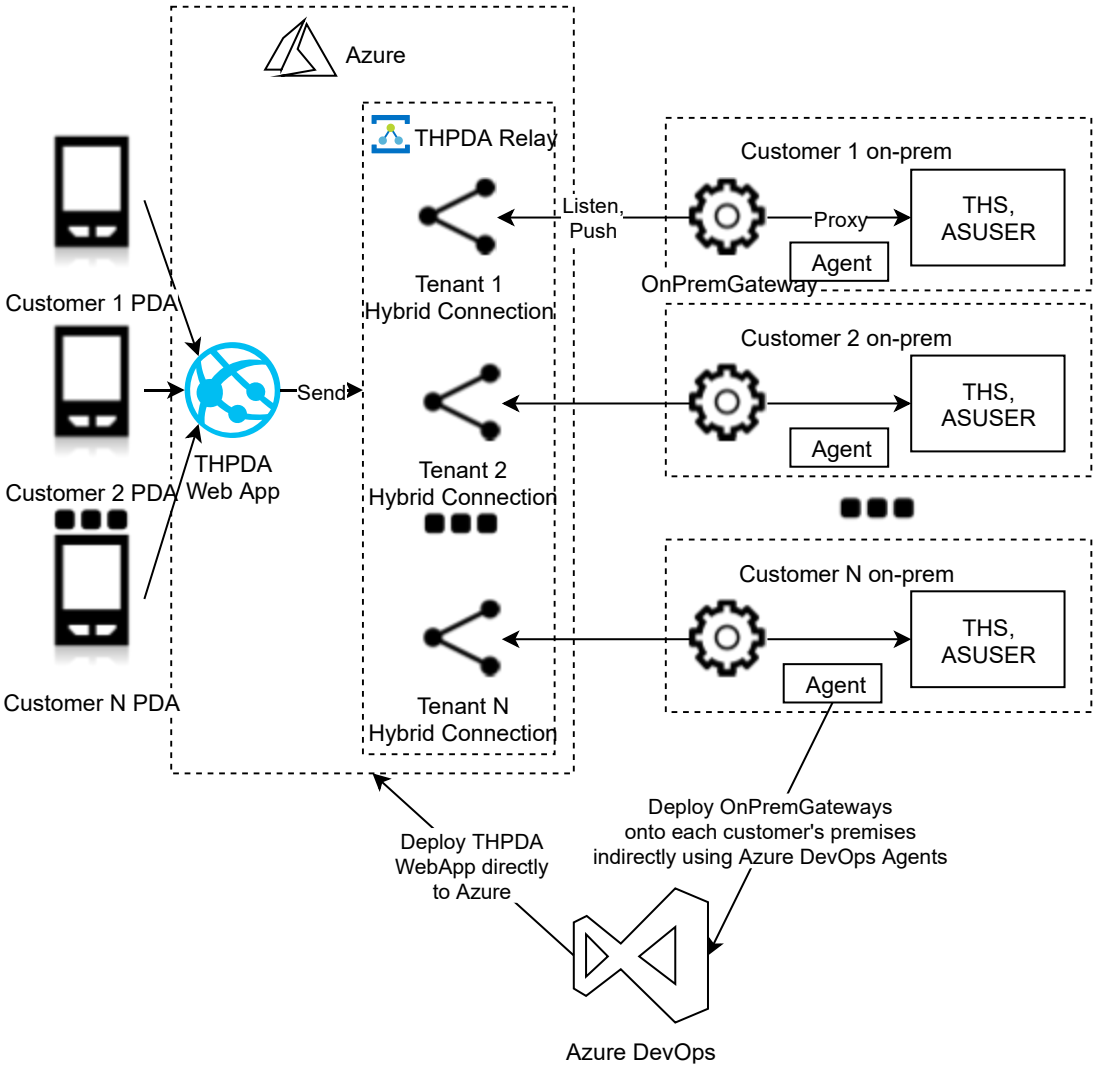

Figure 19. *The overall THPDA MTHC architecture. Vehicle maintenance personnel use PDA devices to access THPDA Web App. THPDA Web App is indirectly connected to the OnPremGateway instances on each customer's premises via Azure Relay Hybrid Connections. THPDA Web App and the overall Azure infrastructure is deployed directly using Azure DevOps Pipelines. The OnPremGateway instances are deployed indirectly using Azure DevOps Pipelines and the deploy agents installed on-premises.*

Serving all tenants, THPDA Web App requires all tenants' AppConfigs, whereas each tenant's dedicated OnPremGateway requires only tenant-specific AppConfigs. Each tenants' AppConfigs are managed in Azure DevOps, so both of the applications can be built and deployed using Azure Pipelines.

8.2 Evaluation

THPDA was developed with the ReactJS/ASP.NET Core stack and it runs on Azure Web App as a multi-tenant application, so Goals 1 and 4 were met. Considering Goal 2, the

deployment model is completely automated, but it involves the same challenges regarding changing customer infrastructure as DQM. Luckily, since the deploy agents are shared between the two and any other future projects, this challenge, if ever realized, would have to be resolved only once per customer – not once *per project* per customer.

Considering Goal 3, the current deployment model of THPDA does not fully respect the ease of introduction philosophy of multi-tenant SaaS. In order to introduce a new tenant, the AppConfig must be manually extended, and a new version of the application must be built and deployed. This model is enough for the time being but could still be improved. A very trivial improvement would be to use *Azure App Configuration Service* (AACS) for storing tenant configurations. AACS would provide a centralized place for storing tenant specific configuration: instead of THPDA Web App and OnPremGateways consuming configuration directly from configuration files produced build-time, they would consume it from a remote AACS instance. If a new customer subscribed to the application, its configuration settings would be written manually only in AACS and then synchronized to the applications automatically. This would remove the need for a build on tenant subscription. Using AACS would also require very little changes to the existing application logic, as only a new *ASP.NET Core Configuration Provider* would have to be registered. Using AACS is still a trade-off, because having a version-controlled AppConfigs in Azure DevOps provides a version history and makes it trivial to connect a specific configuration version to a specific build and release.

In addition to these configuration steps, the current deployment model requires installing OnPremGateway to the customer environment using the local agent. If a customer environment changes – which is a rare occasion – this installation must be done again by the developers. This manual upkeeping does not scale as the number of tenants and tenant environments increase, so a better model would be to allow tenants themselves to install and uninstall their OnPremGateways as they please (i.e. *self-service governance*).

Being a minimum-viable-product and having very little variability requirements to come anytime soon, *variability over the application lifecycle* was considered a non-issue and the variability model evolution was not given much attention. Still, one point of variability that should have been considered is the method of authentication. The current implementation allows users to use ASUSER as the identity authority, but in the dealership business there are several new ongoing projects that are starting to use the company's homebrew OpenID SSO service for user login. Allowing each customer to choose the method of authentication would require some changes to the application, but for the time being, there has not been an explicit demand to support this.

9. SUMMARY & EVALUATION

This thesis focused on implementing new functionality with new technologies on top of old legacy systems using the multi-tenant hybrid cloud architecture. The aim of this approach was to make the deployment model more scalable and automated, and to make development with modern tooling possible. These aspects would potentially provide cost savings and a more familiar toolset for e.g. any new developers.

First, a literature review was conducted in order to explore the challenges and caveats related to the architectural approach. A classification of design challenges was formed. This classification was to be used as a reference when implementing actual add-on applications based on the multi-tenant hybrid cloud architecture. Surprisingly, very few apt case examples focusing exactly on the multi-tenant hybrid cloud architecture were found. There were plenty of papers focusing either on multi-tenancy or on hybrid cloud deployment, so the classification was formed based on these two separately. The lack of multi-tenant hybrid cloud focused papers would indicate that this subject should be further explored by future studies. Proposals for future studies:

- **Hybrid cloud connectivity:** How to integrate tenants' on-premises resources to the multi-tenant systems securely and with little intrusion while still allowing tenants themselves to have control and responsibility over the connectivity (i.e. allowing them to conduct any installation or uninstallation steps themselves).
- **Multi-tenant hybrid cloud as a cloud migration pattern:** As legacy OTS/ASP systems are migrated to cloud, the multi-tenant hybrid cloud architecture could be used as a migration pattern. For example, a legacy application could be refactored progressively to cloud native microservices using the strangler pattern. It would be extremely beneficial to have a process for applying the strangler pattern in a multi-tenant hybrid cloud context.

The current lack of multi-tenant hybrid cloud literature may have a negative effect on the final classification. An extra iteration of snowballing could have led to discovery of papers with more focus on the exact subject. In addition, the scope of the literature review should have been more focused, as now it covers everything from hybrid cloud connectivity security to multi-tenant feature management. Despite these potential disadvantages, the final classification does offer an applicable set of challenges to consider. It was a beneficial checklist when designing the case example applications. And, even though the breadth of the overall subject was not ideal for the thesis, the two concepts, multi-tenancy

and hybrid cloud, go together regarding the architecture and thus having this approach was beneficial from the business perspective.

The technologies that were considered in the scope of this paper were limited to .NET Core stack and Azure services. This was due to both organizational guidelines and to keep the study more focused. Other cloud service providers may have some tooling in their providing that could be even more beneficial regarding multi-tenant hybrid cloud architecture. This also meant that some popular hybrid connectivity enabling solutions were left out of the scope of this paper, such as OpenVPN and ngrok.

Next, two case examples were presented. In these case examples, the multi-tenant hybrid cloud architecture was explored as a potential solution model. The first case example presented the design process of a DQM application. Due to data confidentiality and connectivity reasons a full multi-tenant hybrid cloud architecture would not have been viable. In the end, the application uses the traditional OTS/ASP deployment model, but, despite that, some development scalability was still able to be achieved by using continuous delivery to customer environments.

The second case example focused on a PDA application for supporting the business process of a tire hotel service. As discussed in chapter 8, the multi-tenant hybrid cloud architecture was chosen for the application. A multi-tenancy model was designed, and a hybrid connectivity technology was chosen.

Considering the goals set for the multi-tenant hybrid cloud architecture, Goal 1 was reached in both projects, as they were both developed with the ReactJS/ASP.NET Core stack. Goal 2 was mostly reached in both projects. The deployment models still suffer from the risk of changing customer infrastructure, which would mean that the customer environment would have to be set up again. The root cause for this challenge, lack of self-service governance capabilities, made Goal 3 only partially reachable in both projects, as subscription requires certain manual efforts for setting up the customer environment. Because DQM was decided to be an OTS/ASP deployment and THPDA was decided to be run as a multi-tenant Azure App Service deployment, Goal 4 was only reached by THPDA.

Instead of requiring developers to set up each new customer's environment – or an existing customer's new environment – self-service governance capabilities could be designed. Ideally, these capabilities would allow customers to perform all customer environment specific setup efforts by themselves. As an example, an installer could be provided to those who administer the customer's environment, which would setup and

launch the local OnPremGateway instance. These administrators could own the customer specific configuration and maintain it, for example, via AACS or via local environment variables. Since, as discussed, OnPremGateway is “merely a proxy”, not containing any business domain specific logic, it practically never has to be redeployed, i.e. it does not benefit much from release automation and the deploy agent. A single installation with an installer would be enough. Because, in this scenario, a deploy agent would not be necessary, and because OnPremGateway is a self-contained application, there are very few customer environment prerequisites left in order to get OnPremGateway up and running. The installer could setup OnPremGateway as a Windows Service or even as a Linux Daemon. Having as little prerequisites as possible would make the subscription process more straightforward and, in many cases, more desirable.

Even further, OnPremGateway could cover an entire product portfolio instead of just a single product. For example, instead of just supporting THPDA, OnPremGateway could support all dealership multi-tenant hybrid cloud applications. This would make it no longer necessary to implement a separate on-premises gateway for each project case-by-case *and* the customer would have to deal with only one on-premises gateway instance, making duplication of integration processes less of a concern. The product portfolio point of view was not considered in the scope of this thesis, but surely is something that should be explored more in the future.

The discovered list of multi-tenant hybrid cloud design challenges should be assessed case by case. There were some design challenges that were not covered by either of these cases. Neither of the add-on applications had a storage of their own, so storage variability was not an issue. The variability requirements for both applications were quite low, so no complex multi-tenant configurability models had to be designed. In the oncoming projects with multi-tenant hybrid cloud as a potential approach, these uncovered challenges must be given extra attention.

REFERENCES

[1] ASP.NET Boilerplate, Multi-Tenancy, [Online], Available (accessed 15.9.2019): https://aspnetboilerplate.com/Pages/Documents/Multi-Tenancy

[2] M. Babar, L. Chen, F. Shull, Managing Variability in Software Product Lines, IEEE Software, Vol. 27, Issue 3, May-June, 2010, pp. 89-91.

[3] C.-P. Bezemer, A. Zaidman, Multi-tenant SaaS applications: Maintenance dream or nightmare?, Proceedings of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution, New York, USA, 2010, pp. 88-92.

[4] G. Breiter, V.K. Naik, A framework for controlling and managing hybrid cloud service integration, Proceedings of the 1st IEEE International Conference on Cloud Engineering, San Francisco, California, United States, March 25.-28., 2013.

[5] J.G. Chang, W. Sun, Y. Huang, H.W. Zhi, B. Gao, A framework for native multi-tenancy application development and management, Proceedings of the 9th IEEE International Conference on E-Commerce Technology, Tokyo, Japan, July 23.-26., 2007, pp. 551-558

[6] S. Chen, Y. Qian, X. Zang, R. Peng, A Proxy Based Connection Mechanism for Hybrid Cloud Virtual Network, Proceedings of the 3rd IEEE International Conference on Big Data Security on Cloud, 3rd IEEE International Conference on High Performance and Smart Computing and 2nd IEEE International Conference on Intelligent Data and Security, Beijing, China, May 26.-28., 2017, pp. 80-85.

[7] S.K.S. Cheung, Hybrid cloud deployment of an ERP-based student administration system, 17th Asia-Pacific Web Conference, Guangzhou, China, September 18.-20., 2015.

[8] F. Chong, G. Carraro, Building Distributed Applications: Architecture Strategies for Catching the Long Tail, MSDN Library, 2006.

[9] cloudscribe, Multi-Tenant Support, [Online], Available (accessed 15.9.2019): https://www.cloudscribe.com/multi-tenant-support

[10] A. Correia, J.R. Penha, A.M.R. Da Cruz, An architectural model for customizing the business logic of SaaS applications, Proceedings of the 8th International Joint Conference on Software Technologies, Reykjavik, Iceland, July 29.-31., 2013, pp. 162-268.

[11] K. Costello. Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019, Gartner, [Online], Available (accessed 8.9.2019): https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g

[12] M. Decat, J. Bogaerts, B. Lagaisse, W. Joosen, Amusa: Middleware for efficient access control management of multi-tenant SaaS applications, Proceedings of the ACM Symposium on Applied Computing, Salamanca, Spain, April 13.-17., 2015, pp. 2141-2148.

[13] European Comission, What does the General Data Protection Regulation (GDPR) govern? [Online], Available (accessed 15.9.2019): https://ec.europa.eu/info/law/law-topic/data-protection/reform/what-does-general-data-protection-regulation-gdpr-govern_en

[14] F. Faul, R. Arizcorreta, F. Dudouet, T.M. Bohnert, Application splitting in the cloud: A performance study, Proceedings of the 6th International Conference on Cloud Computing and Services Science, Rome, Italy, April 23.-25., 2016, pp. 245-252.

[15] Finbuckle.MultiTenant, Getting Started, [Online], Available (accessed 15.9.2019): https://www.finbuckle.com/MultiTenant/Docs/GettingStarted

[16] Finbuckle.MultiTenant, MultiTenant Stores, [Online], Available (accessed 15.9.2019): https://www.finbuckle.com/MultiTenant/Docs/Stores

[17] F.S. Foping, I.M. Dokas, J. Feehan, S. Imran, A new hybrid schema-sharing technique for multitenant applications, 4th International Conference on Digital Information Management, Ann Arbor, Michigan, United States, November 1.-4., 2009, pp. 210-215.

[18] U. Ganguly, C. Ray, A secured and cost-effective method for processing queries using cloud resources optimally at hybrid cloud, International Conference on Computer, Electrical and Communication Engineering, Kolkata, India, December 16.-17., 2017, Article number 8009566.

[19] M.F. Gholami, F. Daneshgar, G. Beydoun, F. Rabhi, Key challenges during legacy software system migration to cloud computing platforms – an empirical study, Information Systems, Vol. 67, 2017, pp. 100-113.

[20] M. Hajjat, X. Sun, Y.-W.E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, M. Tawarmalani, Cloudware bound: Planning for beneficial migration of enterprise applications to the cloud, Proceedings of the ACM SIGCOMM 2010 conference, New Delhi, India, August 30. – September 3., 2010, pp. 243-254.

[21] H. Hinton, Cyber Security and Threats: Concepts, Methodologies, Tools and Applications, IBM Coproration, United States, 2018, pp. 102-131.

[22] J.-M. Horcas, M. Pinto, L. Fuentes, Product Line Architecture for Automatic Evolution of Multi-Tenant Applications, 2016 IEEE 20th International Enterprise Distributed Object Computing Conference (EDOC), Vienna, Austria, September 5.-9., 2016, pp. 99-108.

[23] IBM, Mainframe Concepts: IBM, 2005, [Online], Available (accessed 25.8.2019): https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zmainframe/zmainframe_book.pdf

[24] IdentityServer4, The Big Picture, [Online], Available (accessed 8.9.2019): http://docs.identityserver.io/en/latest/intro/big_picture.html

[25] T. Jastrow, T. Preuss, The Entity-Attribute-Value Data Model in a Multi-tenant Shared Data Environment, Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Krakow, Poland, November 4.-6., 2015, pp. 494-497.

[26] M. Jones, J. Bradley, N. Sakimura, RFC 7519: JSON Web Token (JWT), 2015, [Online], Available (accessed 25.8.2019): https://tools.ietf.org/html/rfc7519

[27] J. Kabbedjik, C.-P. Bezemer, S. Jansen, A. Zaidman, Defining Multi-Tenancy: A Systematic Mapping Study on the Academic and the Industrial Perspective, Journal of Systems and Software, Vol. 100, 2015, pp. 139-148.

[28] T. Karthikeyan, T. Nandhini, Dependent component cost model of legacy application for hybrid cloud, Proceedings of IEEE International Conference on Circuit, Power and Computing Technologies, Nagercoil, India, March 18.-19., 2016, Article number 7530154.

[29] A. Leff, J.T. Rayfield, Integrator: An Architecture for an Integrated Cloud/On-Premise Data-Service, Proceedings of the IEEE International Conference on Web Services, New York, United States, June 27. through July 2., 2015, pp. 98-104.

[30] P.-J. Maenhaut, H. Moens, V. Ongenae, F. De Turck, Scalable user data management in multi-tenant cloud environments, Proceedings of the 10th International Conference on Network and Service Management, Rio de Janeiro, Brazil, November 17.-21., 2014, pp. 268-271.

[31] P. Mell, T. Grance. The NIST Definition of Cloud Computing. USA: National Institute of Standards and Technology, 2011.

[32] Microsoft, App Service overview, [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/en-us/azure/app-service/overview

[33] Microsoft, ASP.NET Core fundamentals, [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/fi-fi/aspnet/core/fundamentals/?view=aspnetcore-2.2

[34] Microsoft, ASP.NET Core Middleware, [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.2

[35] Microsoft, Azure App Service Hybrid Connections, [Online], Available (accessed 1.9.2019): https://docs.microsoft.com/en-us/azure/app-service/app-service-hybrid-connections

[36] Microsoft, Azure App Service plan overview, [Online], Available (accessed 8.9.2019) https://docs.microsoft.com/en-us/azure/app-service/overview-hosting-plans

[37] Microsoft, Azure Cosmos DB and multi-tenant systems, [Online], Available (accessed 20.10.2019): https://azure.microsoft.com/en-au/blog/azure-cosmos-db-and-multi-tenant-systems/

[38] Microsoft, Azure Pipelines agent: Windows System Prerequisites, [Online], Available (accessed 1.9.2019): https://github.com/microsoft/azure-pipelines-agent/blob/master/docs/start/envwin.md

[39] Microsoft, Azure Pipelines agents, [Online], Available (accessed 1.9.2019): https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/agents?view=azure-devops

[40] Microsoft, Best practices for securing PaaS web and mobile applications using Azure App Service, [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/en-us/azure/security/fundamentals/paas-applications-using-app-services

[41] Microsoft, Dependency injection in ASP.NET Core, [Online], Available (accessed 1.9.2019): https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2

[42] Microsoft, Deployment groups, [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/en-us/azure/devops/pipelines/release/deployment-groups/?view=azure-devops

[43] Microsoft, Global Query Filters, [Online], Available (accessed 22.9.2019): https://docs.microsoft.com/en-us/ef/core/querying/filters

[44] Microsoft, Inbound and outbound IP addresses in Azure App Service, [Online], Available (accessed 1.9.2019): https://docs.microsoft.com/en-us/azure/app-service/overview-inbound-outbound-ips

[45] Microsoft, Introduction to Identity on ASP.NET Core, [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.2

[46] Microsoft, .NET Core application deployment, [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/en-us/dotnet/core/deploying/

[47] Microsoft, ODBC Overview, [Online], Available (accessed 22.9.2019): https://docs.microsoft.com/en-us/sql/odbc/reference/odbc-overview

[48] Microsoft, What are public, private and hybrid clouds? [Online], Available (accessed 1.9.2019): https://azure.microsoft.com/en-gb/overview/what-are-private-public-hybrid-clouds/

[49] Microsoft, What is ASP.NET Core? [Online], Available (accessed 1.9.2019): https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core

[50] Microsoft, What is Azure Pipelines? [Online], Available (accessed 8.9.2019): https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops

[51] Microsoft, What is Azure Relay? [Online], Available (accessed on 1.9.2019): https://docs.microsoft.com/fi-fi/azure/service-bus-relay/relay-what-is-it

[52] Microsoft, What is Azure Virtual Network? [Online], Available (accessed 1.9.2019): https://docs.microsoft.com/bs-latn-ba/azure/virtual-network/virtual-networks-overview

[53] Microsoft, What is IaaS?: Infrastructure as a service, [Online], Available (accessed 8.9.2019): https://azure.microsoft.com/en-us/overview/what-is-iaas/

[54] Microsoft, What is PaaS?: Platform as a service, [Online], Available (accessed 8.9.2019): https://azure.microsoft.com/en-us/overview/what-is-paas/

[55] Microsoft, What is SaaS?: Software as a service, [Online], Available (accessed 8.9.2019): https://azure.microsoft.com/en-us/overview/what-is-saas/

[56] Microsoft, What is VPN Gateway? [Online], Available (accessed 1.9.2019): https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpn-gateways

[57] C. Miyachi, What is "Cloud"? It is time to update the NIST definition? IEEE Cloud Computing, Vol. 5, Issue 3, June 2018, pp. 6-11.

[58] F. Mohamed, M. Abu-Matar, R. Mizouni, M. Al-Qutayri, Z.A. Mahmoud, SaaS dynamic evolution based on model-driven software product lines, Proceedings of the International Conference on Cloud Computing Technology and Science, Singapore, Singapore, 2015, pp. 292-299.

[59] E. Moyle, Legacy application migration to the cloud and security, [Online], Available (accessed 25.8.2019): https://searchcloudsecurity.techtarget.com/tip/Legacyapplication-migration-to-the-cloud-and-security

[60] OpenID, What is OpenID Connect? [Online], Available (accessed 1.9.2019): https://openid.net/connect/

[61] C. Pahl, H. Xiong, R. Walshe, A comparison of on-premise to cloud migration approaches, Proceedings of European Conference on Service-Oriented and Cloud Computing ESOCC, 2013, pp. 212-216

[62] R.C. Pathak, P. Khandelwal, A model for hybrid cloud integration: With a case study for IT service management (ITSM), Proceedings of the 6th IEEE International Conference on Cloud Computing in Emerging Markets, Bangalore, India, November 1.-3., 2017, pp. 113-118.

[63] K. Petersen, R. Feldt, S. Mujtaba, M. Mattson, Systematic Mapping Studies in Software Engineering, 12th International Conference on Evaluation and Assesment in Software Engineering, Italy, 2008, 9 p.

[64] M. Rouse, Definition: IBM i, [Online], Available (accessed 25.8.2019): https://whatis.techtarget.com/definition/IBM-i

[65] M.D. Samrajesh, N.P. Gopalan, Towards Multivariable Architecture for SaaS Multi-tenant Applications, International Journal of Software Engineering and Its Applications, Vol. 10, No. 4, 2016, pp. 13-26.

[66] R.K. Shyamasundar, N.V.N. Kumar, M. Rajarajan, Information-flow control for building security and privacy preserving hybrid clouds, Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications, 14th IEEE International Conference on Smart City and 2nd IEEE Conference on Data Science and Systems, Sydney, Australia, December 12.-14., 2016, pp. 1410-1417.

[67] E. Sturrus, O. Kulikova, Orchestrating hybrid cloud deployment: An overview, Vol. 47, Issue 6, June, 2014, pp. 85-87.

[68] W. Su, C. Lin, K. Meng, Q. Liu, Modeling and analysis of availability for SaaS multi-tenant architecture, Proceesings of IEEE 8th International Symposium on Service Oriented System Engineering, Oxford, United Kingdom, April 7.-11., 2014, pp. 365-369.

[69] A.N. Toosi, R. Buyya, Research Advances in Cloud Computing, Springer, Melbourne, Australia, 2017, pp. 93-114.

[70] E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, T. D'Hont, W. Joosen, Context-oriented programming for customizable SaaS applications, 27th Annyal ACM Symposium on Applied Computing, March 26.-30., 2012, pp. 418-425.

[71] D. Van Landuyt, G. Fatih, E. Truyen, W. Joosen, Middleware for Dynamic Upgrade Activation and Compensations in Multi-tenant SaaS, Proceedings of the 15th International Conference on Service-Oriented Computing, Malaga, Spain, November 13.-16., 2017, Springer, pp. 340-348.

[72] S. Walraven, E. Truyen, W. Joosen, A middleware layer for flexible and cost-efficient multi-tenant applications, 12th ACM/IFIP/USENIX International Middleware Conference, Lisbon, Portugal, December 12.-16., 2012, pp. 370-389.

[73] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, W. Joosen, Efficient customization of multi-tenant Software-as-a-Service applications with service lines, Vol. 91, Issue 1, May, 2014, pp. 48-62.

[74] H. Yaish, M. Goyal, A multi-tenant database architecture design for software applications, Proceedings of the 16th IEEE International Conference on Computational Science and Engineering, Sydney, Australia, December 3.-5., 2013, pp. 933-940.

[75] H. Yaish, M. Goyal, Multi-tenant database access control, Proceedings of the 16th IEEE International Conference on Computational Science and Engineering, Sydney, Australia, December 3.-5., 2013, pp. 870-877.

[76] J. Zeng, B. Plale, Argus: A Multi-tenancy NoSQL store with workload-aware resource reservation, Parallel Computing: Systems & Applications, Vol. 58, October, 2016, pp. 76-89.

[77] C. Zou, H. Deng, Design and implementation of hybrid cloud computing architecture based on cloud bus, Proceedings of the 9th International Conference on Mobile Ad-Hoc and Sensor Networks, Dailan, China, December 11.-13., 2013, pp. 289-293.